TURING 图灵程序设计丛书

Packt›

Hands-on
Natural Language Processing with Python

# Python 自然语言处理实战

[印] 拉杰什·阿鲁姆甘　拉贾林加帕·尚穆加马尼◎著　　杨航◎译

- 全面讲解NLP前沿技术，构建多种实用应用程序
- 运用实际数据集和流行库，图示丰富、代码清晰易懂


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

作者

**拉杰什·阿鲁姆甘**
（Rajesh Arumugam）

目前在新加坡SAP公司负责机器学习开发工作，此前曾与日立亚洲（新加坡）社会创新中心合作，为智慧城市的多个领域开发过机器学习解决方案。毕业于南洋理工大学，获计算机工程博士学位，曾在多个会议上发表过论文，并在存储和机器学习方面拥有专利。

**拉贾林加帕·尚穆加马尼**
（Rajalingappaa Shanmugamani）

目前在Kairos担任技术经理，此前作为数据学习专家在新加坡SAP公司创新中心工作，并在开发计算机视觉产品的许多创业公司负责过开发和咨询工作。毕业于印度理工学院马德拉斯分校，获硕士学位，学位论文主题基于产业中计算机视觉的应用程序。他还在该领域发表了若干论文。

# 数字版权声明



图灵程序设计丛书

Hands-on Natural Language Processing with Python

# Python自然语言处理实战

[印] 拉杰什·阿鲁姆甘　拉贾林加帕·尚穆加马尼◎著　　杨航◎译

人民邮电出版社
北　京

图书在版编目（CIP）数据

Python自然语言处理实战 / (印) 拉杰什·阿鲁姆甘, (印) 拉贾林加帕·尚穆加马尼著 ; 杨航译. -- 北京 : 人民邮电出版社, 2020.10
（图灵程序设计丛书）
ISBN 978-7-115-54926-6

Ⅰ. ①P… Ⅱ. ①拉… ②拉… ③杨… Ⅲ. ①软件工具—程序设计②自然语言处理 Ⅳ. ①TP311.561②TP391

中国版本图书馆CIP数据核字(2020)第181557号

## 内 容 提 要

本书介绍自然语言处理和深度学习的核心概念，例如CNN、RNN、语义嵌入和Word2vec等。读者将学习如何使用神经网络执行自然语言处理任务，以及如何在自然语言处理应用程序中训练和部署神经网络。读者会在各种应用领域中使用RNN和CNN，例如文本分类和序列标记，这对于情绪分析、客服聊天机器人和异常检测的应用至关重要。读者还将掌握使用Python流行的深度学习库TensorFlow在语言应用程序中实现深度学习的实用知识。

希望利用自然语言处理技术构建深度学习应用的Python开发人员、机器学习或自然语言处理工程师都可以从本书中获益。

◆ 著　[印] 拉杰什·阿鲁姆甘　拉贾林加帕·尚穆加马尼
译　杨　航
责任编辑　杨　琳
责任印制　周昇亮
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　印刷
◆ 开本：800×1000　1/16
印张：14.25
字数：337千字　2020年10月第1版
印数：1－2 500册　2020年10月北京第1次印刷
著作权合同登记号　图字：01-2019-3977号

定价：59.00元

**读者服务热线：(010)51095183转600　印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京东市监广登字20170147号**

# 版 权 声 明

# 序

目前，语音转录、机器翻译、对话代理和情感分析等形式的智能数字助理已被广泛应用于各个领域，让人机交互愈发便利。聊天机器人正在成为多数网站的组成部分，虚拟助手在家庭和办公室中也越来越受欢迎。虽然有大量资源介绍了**自然语言处理**（natural language processing，NLP）概念下的这些主题，但本书不仅涵盖了相关基础知识和最新技术，还包括流行框架和工具包的使用示例，是一本 NLP 方面不可多得的全面指南。

初次受邀为本书作序时，我很高兴可以传达出驱使作者写作本书的那份热情，但不确定如何才能最好地展示这一绝佳的最新知识来源，以及能真正脱颖而出、用于 NLP 的**机器学习**（machine learning，ML）实用手册。

本书作者在机器学习领域的声誉无须过多解释。凭借在世界一流大学中的学术教育水平以及在 ML 开发方面的多年领导经验，拉杰什和拉贾林加帕是撰写本书的绝佳人选。在我眼中，他们不仅学识渊博，而且是热情的教育家，能用简单的语言传达复杂的概念。拉贾林加帕热衷于帮助初创企业起步并以开放的心态向年轻公司提供专业知识，令人钦佩。我相信，即使作为本书的读者，你也可以向他提问，并一定会得到令人信服的答案。

本书结构合理，写得也很出彩。从具体的例子到基本的概念解释、再到代码片段，本书可以指导具有各层次深度学习知识水平的读者。本书的章节结构得到了精心的设计，以保证在整个过程中都能使读者全神贯注。你将兴奋地发现本书将文本处理与分类的流行技术与最新方法结合在了一起。

通过阅读本书，你将学习到如何执行常见的 NLP 任务，例如使用 Python 的自然语言工具包进行文本的预处理和探索性分析；了解深度神经网、Google 的 TensorFlow 框架和**递归神经网络**的构造块，包括长短期记忆网络；还将掌握词嵌入的概念，以便在上下文中使用语义。

讲完基础知识后，本书将进一步指导你开发各种应用架构和深度神经网络模型，包括文本分类、文本生成和文本摘要、问答、语言翻译、语音识别以及文本转语音。

本书最后提供了各种方法来在各个平台上部署训练好的 NLP 任务模型。学完本书，你将了解 NLP 中的数据科学范式，并且能按照作者的设想在生产环境的商业应用中部署深度学习模型。

Maryam Azh 博士<br>Overlay Technologies 创始人

# 前　　言

在深度学习出现之前，传统的**自然语言处理**（natural language processing，NLP）方法已被广泛用于诸如垃圾邮件过滤、情感分类和**词性**（part of speech，POS）标注等任务中。这些经典方法利用了序列的统计学特征（如字数统计和共现）以及简单的语言特征。但是，这些技术的主要缺点在于无法捕获复杂的语言特征，例如上下文和单词依存关系。

神经网络和深度学习的最新发展为我们提供了强大的新工具，能在 NLP 任务中达到人类级别的表现，并构建用于处理自然语言的产品。用于 NLP 的深度学习围绕着词嵌入或词向量（也称为 Word2vec）的概念展开，该概念将单词和短语的含义封装为密集的向量表示形式。与传统的独热编码（one-hot encoding）相比，词向量能够更好地捕获单词的语义信息。把它与**循环神经网络**（recurrent neural network，RNN）结合使用，能以直观的方式处理语言的时序特征。尽管 RNN 只能捕获局部的单词依存关系，但最近有人提出了基于向量对词向量序列执行 attention（注意力）和对齐操作，允许神经网络对包括上下文在内的全局单词依存关系进行建模。由于能够对语言的语法和语义进行模型化，并且具有强大的经验表现以及适应新数据的泛化能力，神经网络已成为构建高度复杂商业产品（例如搜索引擎、翻译服务和对话系统）的首选模型。

本书介绍了 NLP 深度学习模型的基本构建模块，并探讨了最新文献中的前沿技术。我们采用基于问题进行实战学习的方法，在各种 NLP 任务中引入新模型作为解决方案。我们着重于提供用 Python 实现的实用代码，你也可将其应用于自己的用例，为你的应用引入能与人类媲美的能力。

## 目标读者

本书适合希望利用 NLP 技术开发具有丰富人性化接口的智能应用的开发者。本书假定你已具备**机器学习**（machine learning，ML）或深度学习的入门知识，以及中级的 Python 编程技能。我们的目标是使用 TensorFlow 框架和 Python 来介绍用于情感检测、对话系统、语言翻译和语音转文本等 NLP 任务的前沿技术。

你将从深度学习的基本概念出发，逐步深入处理自然语言的最新算法和最佳实践。我们着力于使用实际数据实现应用并部署深度学习模型，以在生产环境中为商业应用赋予与人类媲美的能力。

## 本书内容

**第 1 章“起步”**　这一章探讨了 NLP 的基本概念及其试图解决的各种问题。我们还将介绍一些现实中的应用，使你体会到 NLP 的广泛使用。

**第 2 章“使用 NLTK 进行文本分类和词性标注”**　这一章介绍了流行的 Python 库 NLTK。我们将使用 NLTK 来描述基本的 NLP 任务，例如分词、词干提取、标注和经典文本分类，并将利用 NLTK 来探索词性标注。我们为你提供了必需的工具和技术，用以准备要输入深度学习模型中的数据。

**第 3 章“深度学习和 TensorFlow”**　这一章介绍了深度学习的基本概念，还将帮助你设置环境和工具（如 TensorFlow）。在这章的最后，你将了解基本的深度学习概念，例如卷积神经网络（convolutional neural network，CNN）、循环神经网络（recurrent neural network，RNN）、长短期记忆（long-short term memory，LSTM）网络、基于 attention 的模型以及 NLP 中的问题。

**第 4 章“使用浅层模型进行语义嵌入”**　这一章探讨了如何识别文档中单词间的语义关系，我们会在此过程中获得单词在语料库中的向量表示。这章介绍了使用神经网络开发词嵌入模型，例如连续词袋模型（continuous bag-of-words，CBOW），还对开发神经网络模型以获得文档向量的技术进行了介绍。在这章的最后，你将会熟悉单词、句子和文档的嵌入训练，并实现简单网络的可视化。

**第 5 章“使用 LSTM 进行文本分类”**　这一章讨论了各种文本分类方法，其中一个具体应用是对文档中的单词或短语进行情感分类。这章介绍了文本分类问题，之后描述了使用 CNN 和 LSTM 开发深度学习模型的技术，还介绍了如何使用预训练的词嵌入进行迁移学习来用于文本分类任务。最后，你将熟悉如何实现用于情感分类和垃圾邮件检测的深度学习模型，并将预训练好的词嵌入用于自己的分类任务中。

**第 6 章“使用 CNN 进行搜索和去重”**　这一章涵盖了文档搜索、匹配和去重的问题及其解决方法，介绍了如何开发用于在语料库中搜索文本的深度学习模型。在这章的最后，你将学会实现用于文本搜索和去重的 CNN 深度学习模型。

**第 7 章“使用字符级 LSTM 进行命名实体识别”**　这一章描述了执行**命名实体识别**（named entity recognition，NER，它是信息提取的子任务）以定位和分类文档文本中实体的方法。这章介绍了 NER 问题以及该问题的应用范围，然后解释了基于字符的 LSTM 深度学习模型的实现，该模型能用于识别用标记数据集训练得到的命名实体。

**第 8 章“使用 GRU 进行文本生成和文本摘要”**　这一章介绍了用于生成文本的方法，其中一种方法的扩展可用于从文本数据中创建摘要。之后我们将描述如何实现用于生成文本的深度学习模型，并介绍用于文本摘要的基于门控循环单元（gate recurrent unit，GRU）的深度学习模型。在这章末尾，你将学到如何实现用于文本生成和文本摘要的深度学习模型。

**第 9 章“使用记忆网络完成问答任务和编写聊天机器人”**　这一章介绍了如何训练深度学习模型来回答问题并将其扩展成为聊天机器人，还介绍了问答技术这一问题以及使用深度学习模型构建问答引擎的方法。之后我们将描述如何利用问答引擎来构建能够以对话形式进行问答的聊天机器人。在这章的最后，你将能够实现一个交互式聊天机器人。

**第 10 章“使用基于 attention 的模型进行机器翻译”**　这一章涉及无须学习语法结构即可将文本从一种语言翻译成另一种语言的各种方法。这章介绍了传统的机器翻译方法，例如基于隐马尔可夫模型的方法。之后我们将重点介绍利用 attention 将文本从法语翻译为英语的编码 - 解码模型的实现。在这章的最后，你将能够实现用于文本翻译的深度学习模型。

**第 11 章“使用 DeepSpeech 进行语音识别”**　这一章介绍了语音转文本的问题，作为会话式界面的开始。这章从语音数据的特征提取开始，接下来对 Deep Speech 体系结构进行简要介绍，然后解释了语音转文本的 Deep Speech 架构的详细实现。在这章的末尾，你将具备实现语音转文本深度学习模型的相关知识。

**第 12 章“使用 Tacotron 进行文本转语音”**　这一章介绍了文本转语音的问题，以及利用 Tacotron 模型实现将文本转换为语音的过程。最后，你将熟悉基于 Tacotron 架构的文本转语音模型的实现。

**第 13 章“部署训练好的模型”**　最后这一章介绍了在各种云平台和移动平台上的模型部署。

## 如何充分利用本书

阅读本书的先决条件是具有机器学习或深度学习的基本知识，以及中级的 Python 编程技能，但这些都并非必需。我们会在第 1 章中简要介绍深度学习，并涉及诸如多层感知器、卷积神经网络和循环神经网络之类的主题。如果你了解通用的机器学习概念（例如过拟合和模型正则化）以及经典模型（例如线性回归和随机森林），将有所帮助。在更高级主题的章节中，你可能会遇到深入的代码演练，这需要基本的 Python 编程经验。

如下一节所述，本书中的所有示例代码都可以从代码库中下载。这些示例主要利用开源工具和开源数据库，并且都使用 Python 3.5 或更高版本编写。本书中使用最多的库是 TensorFlow 和 NLTK，其详细安装说明分别包含在第 2 章和第 3 章中。尽管运行本书中的示例不需要图形处理器（graphics processing unit，GPU），但建议读者使用包含 GPU 的系统环境，因为更为复杂的任务涉及更大的模型及数据集，所以我们建议在本书的后半部分使用 GPU 进行模型的训练。

### 下载示例代码文件

你可以通过www.packtpub.com上的账户下载本书的示例代码文件。如果是从其他地方购买了本书，则可以访问www.packtpub.com/support并注册，以便我们将文件通过电子邮件发送给你。

你可以通过以下步骤下载代码文件：

(1) 在www.packtpub.com上登录或注册；
(2) 选择**支持**（Support）标签；
(3) 单击**代码下载及勘误**（Code Downloads & Errata）；
(4) 在**搜索**（Search）栏键入书名并遵循屏幕上的指示。

下载好文件之后，请确保使用以下最新版本的文件解压缩/提取工具：

- WinRAR/7-Zip（Windows 系统）;
- Zipeg/iZip/UnRarX（macOS 系统）;
- 7-Zip/PeaZip（Linux 系统）。

## 下载彩色图像

我们还提供了包含本书截图/图表彩色图像的 PDF 文件以供下载。①

## 排版约定

本书会使用如下排版约定。

等宽字体表示代码，如下所示：“`pip` 安装程序可用于安装 NLTK 库，并同时安装 NumPy 库（可选）。”

代码段以如下方式展示：

```
>>> large_words = dict([(k,v) for k,v in frequency_dist.items() if
len(k)>3])
>>> frequency_dist = nltk.FreqDist(large_words)
>>> frequency_dist.plot(50,cumulative=False)
```

命令行输入输出以如下方式展示：

```
import nltk
nltk.download()
```

**黑体**表示新术语和重要词语，如下所示：“导航到**停用词**并安装它，以备将来使用。”



此图标表示警告或者重要注释。

① 代码文件和彩色图像均可以在图灵社区本书主页（ituring.cn/book/2679）下载。——编者注



此图标表示提示和小技巧。

## 联系我们

我们总是欢迎来自读者的反馈。

**一般反馈**：请将邮件发送到feedback@packtpub.com并在主题中提到书名。如果你有本书领域的相关问题，请发送邮件到questions@packtpub.com。

**勘误**：尽管我们已尽最大努力确保本书内容的准确性，但错误总是难以避免的。如果你发现了本书中的错误并将其报告给我们，我们会非常感激。请访问 http://www.packtpub.com/submit-errata，选择图书，单击勘误提交表的链接并输入详细信息。

**反盗版**：如果你在网络上遇到任意形式的非法副本时将地址或网站名称提供给我们，我们将非常感激。请通过copyright@packtpub.com联系我们并附上材料链接。

**成为作者**：如果有你擅长的领域且有意向对此编写图书或做出贡献，请访问 http://authors.packtpub.com/。

## 评论

请留下评论。当你看完或使用完本书后，为什么不在购买网站上留下评论呢？潜在的读者可以看到并利用你的公正观点做出购买选择，在 Packt 的我们可以了解到你关于我们产品的想法，我们的作者也可以看到你的反馈。谢谢你！

如需有关 Packt 的更多信息，请访问https://www.packtpub.com/。

## 电子书

扫描如下二维码，即可购买本书电子版。



## 链接

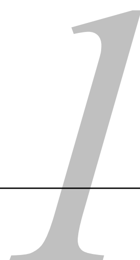
扫描如下二维码，即可获取本书所需网站链接。



# 目　　录

第 1 章

# 起　　步



**自然语言处理**（NLP）是使用计算机来理解人类语言的领域，涉及使用计算机分析大量自然语言数据，以收集数据的意义和价值供实际应用使用。尽管 NLP 自 20 世纪 50 年代就已经存在，但随着机器学习和深度学习的最近发展，该领域的实际应用才有了巨大进展。本书的大部分内容将重点研究 NLP 的各种实际应用（如**文本分类**）以及 NLP 的子任务（例如**命名实体识别**），并将特别着重于深度学习方法。本章将首先介绍 NLP 领域中的基本概念和术语，并在这之后讨论 NLP 技术的一些当前应用。

## 1.1　NLP 中的基本概念和术语

以下是 NLP 中主要与语言数据相关的重要术语和概念。熟悉它们将有助于你速理解本书后续章节中的内容：

- 文本语料库
- 段落
- 句子
- 短语和单词
- *n* 元语法（*n*-gram）
- 词袋（bag-of-words）

我们将会在接下来几节里进行解释。

### 1.1.1　文本语料库

所有 NLP 任务所依赖的语言数据称为文本语料库，简称为语料库。语料库是使用英语、法语等语言的一大组文本数据，可以包含一个或一堆文档。文本语料库的来源可以是 Twitter 等社交网站、博客站点、Stack Overflow 等开放式论坛和图书，等等。在诸如机器翻译之类的某些任务中，我们将需要一个多语种的语料库。例如，我们需要同一个文档内容的英语和法语翻译以开发机器翻译模型。对于语音任务来说，我们还需要人工录音和相应的转录语料库。

在后面的大多数章节里，我们将使用网上或开源数据仓库中的文本语料库和语音录音。对许多 NLP 任务来说，语料库被划分成块以进一步分析，这些分块可以是段落级、句子级或单词级的。我们将在随后几小节接触到它们。

### 1.1.2 段落

段落是NLP任务处理的最大文本单位。除非被细分成句子，否则段落级别的边界本身可能用处不大（尽管有时段落也能被视为上下文的分界）。部分Python库提供了可将文档拆分为多个段落的分词器（tokenizer）[①]。我们将在后续章节对分词器进行介绍。

### 1.1.3 句子

句子是语言数据词法单位的下一层次。句子封装了完整的含义或思想和上下文。它通常是根据由标点符号（如句号）确定的边界从段落中被提取出来。句子还可以传达其中所表达的观点或情感。通常而言，句子由词性（POS）实体（如名词、动词和形容词等）组成。分词器可根据标点符号将段落拆分为句子。

### 1.1.4 短语和单词

短语是句子中可以传达特定含义的一组连续单词。例如，在句子“明天将会是下雨天”中，“将会是下雨天”表达了特定的思想。一些 NLP 任务从句子中提取关键短语用以搜索和检索应用。文本的下一个单位是单词，它也是最小的单位。常见的分词器根据空格和逗号等标点将句子分为单词。当我们在不同的上下文中使用同一个单词时，单词会有歧义，这也是 NLP 的问题之一。稍后我们将在讨论词嵌入时看到如何很好地处理它。

### 1.1.5 *n*元语法

一系列字符或单词构成一个 *n*-gram。例如，一元（unigram）由单个字符组成，二元（bigram）由两个字符的序列组成，依此类推。类似地，单词 *n*-gram 由 *n* 个单词的序列组成。在 NLP 中，*n*-gram 被用作诸如文本分类之类的任务的特征。

### 1.1.6 词袋

与 *n*-gram 相比，词袋不考虑词序，而是捕获文本语料库中单词出现的频数。词袋还被用作情感分析和主题识别等任务之中的特征。

① 部分分词器具有分句功能，所以这里统称分词器。——译者注

在下面各节中，我们将概述 NLP 的以下应用：

- 情感分析
- 命名实体识别
- 实体链接
- 文本翻译
- 自然语言推理
- 语义角色标记
- 关系提取
- SQL 查询生成或语义解析
- 机器阅读理解
- 文字蕴含
- 指代消解
- 搜索
- 问答和聊天机器人
- 文本转语音
- 语音转文本
- 说话人识别
- 口语对话系统
- 其他应用

## 1.2　NLP 技术的应用

本节将概述 NLP 技术的主要应用。此处列出的主题尽管并非详尽，但会使你了解到 NLP 技术的广泛应用。

### 1.2.1　情感分析

句子或文本中的情感反映了产生或怀有该情感的人总体正面、负面或中立的观点或想法。它表示出一个人对于描述文本的主题或上下文是否感到快乐、不快乐或平静。情感可以被量化为离散值，例如 1 表示快乐、–1 表示不快乐、0 表示平静，也可以在 0 ~ 1 连续区间上进行量化。因此，情感分析是从不同数据源（如社交网络、产品评论、新闻文章等）获得的一段文本中得出量化值的过程。情感分析在现实世界中的一种应用是从社交网络数据中得到有价值的情报，例如客户满意度、产品或品牌的知名度、时尚趋势等。图 1-1 显示了情感分析的一种应用，它可以捕获关于 Google 的特定新闻文章的整体意见。

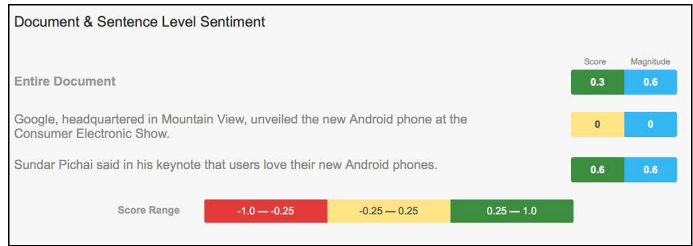



图　1-1

图 1-1 表明，整个文档以及单个句子级别的情感数据都已被捕获。

## 1.2.2　命名实体识别

**命名实体识别**（named entity recognition，NER）是一种文本注释任务。在 NER 中，文本中的单词或词元被解析或注释为具体类别，例如组织、位置和人物等。实际上，NER 将非结构化文本数据转换为结构化数据以便展开进一步分析。图 1-2 是从 Google Cloud API 获得的可视化效果，你可以尝试使用该接口。

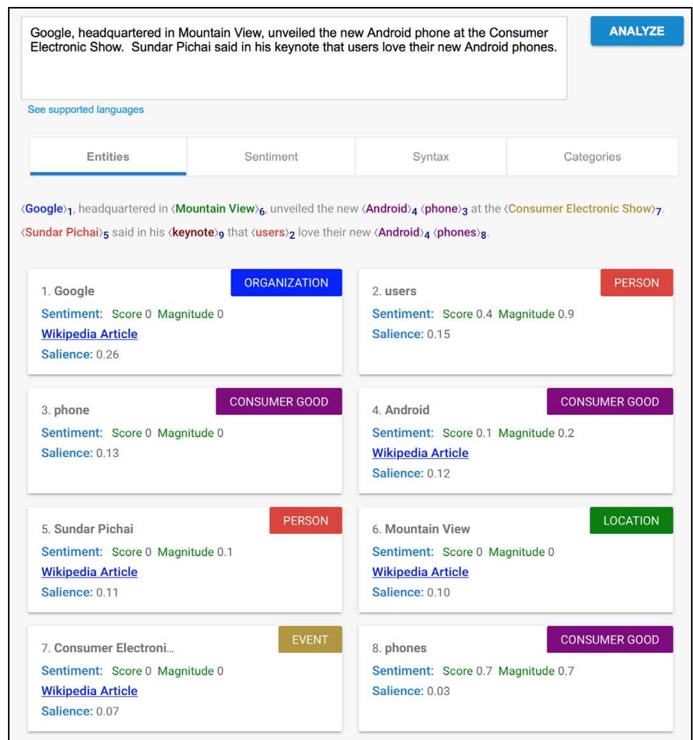



图　1-2

图 1-2 显示了 NER 如何自动地从原始的非结构化文本中提取不同实体，例如**组织**（Google）、**人物**（Sundar Pitchai）和**事件**（**消费电子科技展**），等等。输出还会基于情感分析结果给出每个标签或类别的情感。你可以使用 Google Cloud 尝试不同的文本。当我们单击“**类别**”（Categories）选项卡时，可以看到如图 1-3 所示内容。

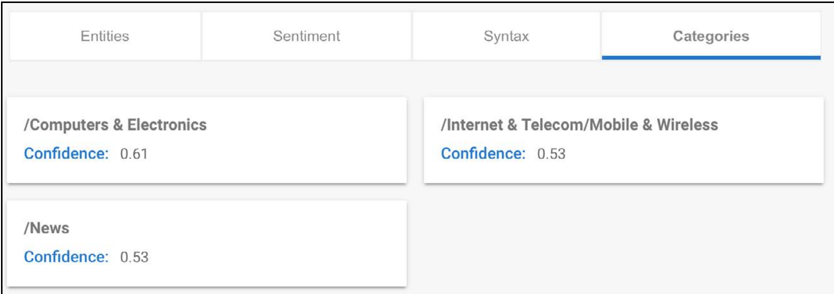



图 1-3

图 1-3 显示了系统如何使用文本的命名实体识别将特定文本分类为**计算机和电子学**、**新闻**等。这种分类称为**主题建模**，是用于识别句子或文档主旨或主题的另一个重要的 NLP 任务。

### 1.2.3 实体链接

NLP 的另一个实际应用是实体链接。我们可以在 Microsoft Azure 的文本分析 API 中找到一个很好的例子。图 1-4 显示了示例文本的输出。

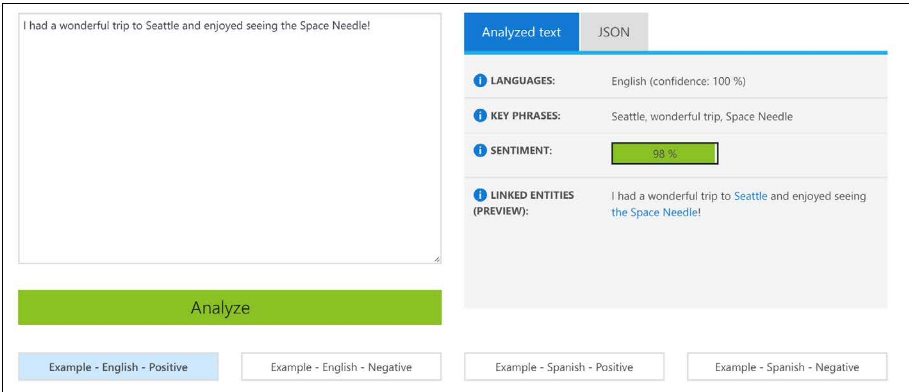



图 1-4

图 1-4 显示了系统是如何自动将实体**西雅图**（Seattle）作为位置提取的。有趣的是，系统通过将**太空针塔**（the Space Needle）与西雅图相连接，正确地将其提取为地标性建筑。这表明在提取实体间有用关系时，命名实体链接的功能是非常强大的。

## 1.2.4　文本翻译

机器翻译将给定的一段文本从一种语言翻译成另一种目标语言。任务中首先确定初始语言，然后将其翻译为目标语言。事实证明，Google 的翻译应用消除了语言障碍，在旅行中非常有用。文本翻译的最新技术大大提高了翻译的准确性。

图 1-5 是访问 Google Translate 并将中文翻译成英文的示例。

Source Language
Chinese (Simplified) (zh)
Target Language
English (en)

Sample text. Enter your own text to translate.
古代書面汉语称为文言文，现代书面汉语一般指使用現代標準漢語語法、詞彙的中文通行文体（又称白话文）。

Ancient written Chinese is called classical Chinese. Modern written Chinese generally refers to the use of modern standard Chinese grammar and Chinese vocabulary (also known as vernacular Chinese).

Translate Text

Request URL

https://translation.googleapis

JSON Response


```
{
  "data": {
    "translations": [
      {
        "translatedText": "Ancient written Chinese is called classical Chinese. Modern written Chinese generally refers to the use of modern standard Chinese grammar and Chinese vocabulary (also known as vernacular Chinese)."
      }
    ]
  }
}
```


图　1-5

图 1-5 还显示了当我们使用 Google 的 Translation API 服务时，翻译文本的 JSON 响应。

## 1.2.5　自然语言推理

**自然语言推理**（natural language inference，NLI）任务对前提和假设之间的关系进行分类。在推理过程中，任务将前提和假设作为输入，并基于给定的前提输出假设是否为真的判断。

## 1.2.6　语义角色标记

**语义角色标记**（semantic role labeling，SRL）能确定给定句子与谓语（例如动词）之间的关系。有时推理是通过问题的形式给出的。一个语义角色的例子可能是：某事在何地或何时发生？图 1-6 是一个出自 AllenNLP 的可视化例子。

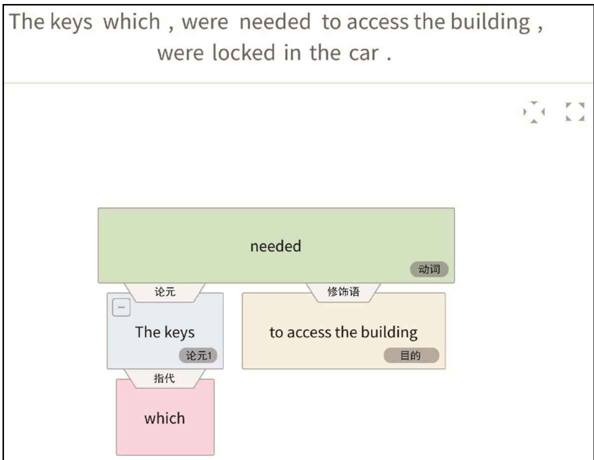



图　1-6

图 1-6 展示了语义标记是如何在不同文本片段之间创建语义关联的。例如，**需要**（needed）**钥匙**（The keys）的目的是**进入大楼**（to access the building）。你可以通过 AllenNLP 去尝试不同的示例。

## 1.2.7　关系提取

当提供有文本和关系类型时，**关系提取**可以进行关系的预测。在某些情况下，关系也可能无法被提取出来。图 1-7 显示了基于谓词和对象的关系提取示例。

TextRazor.　Demo　Technology　Documentation　Pricing　|　Login　Sign up

Edit Text　Language: *eng* Processed in: *0.6672* seconds

**Barclays** misled **shareholders** and the public about one of the biggest **investments** in the **bank**'s history, a **BBC Panorama** investigation has found.

Words　Phrases　Relations　Entities　Meaning　Dependency Parse

| Subject | Predicate | Object |
| --- | --- | --- |
| Barclays | misled about | shareholders and the public one of the biggest investments in the bank history |
| a BBC Panorama investigation | has found | Barclays misled shareholders and the public about one of the biggest investments in the bank history |

| Predicate | Property |
| --- | --- |
| the investments in | biggest |
| the investments in | the bank history |
| the bank | history |

CATEGORIES
0.93 economy, business and finance>economy>macro economics>investments
0.70 economy, business and finance>economy
0.65 economy, business and finance>market and exchange>securities
0.48 economy, business and finance
0.48 economy, business and finance>business information>business finance>shareholder
0.48 crime, law and justice>law
0.47 science and technology>social sciences>economics
0.45 economy, business and finance>market and exchange>loan market>loans
0.43 economy, business and finance>market and exchange
0.43 economy, business and finance>economic sector>financial and business service

图 1-7　关系提取示例

以上示例显示了从示例文本到主题、谓词和对象的关系提取。

## 1.2.8　SQL查询生成或语义解析

语义解析有助于将自然语言转换为 SQL 查询语句以便完成对数据库的查询。图 1-8 展示的示例将自由文本查询转换为 DBpedia 数据库 SPARQL 查询，这与 SQL 非常相似。

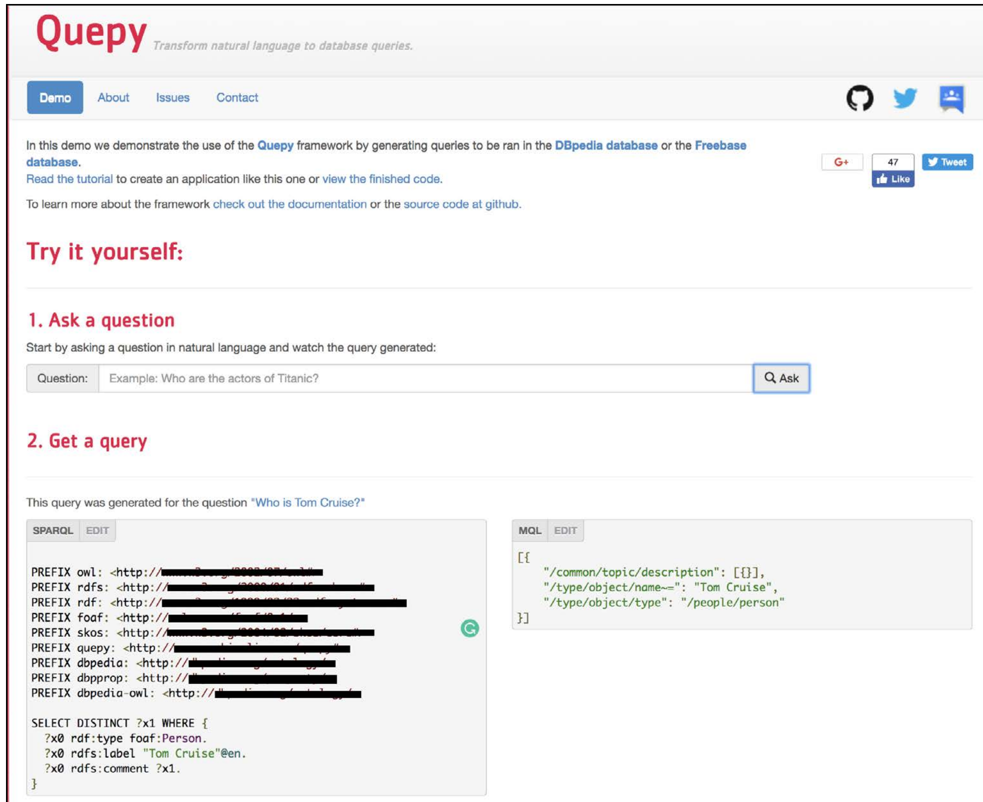



图　1-8

以上可视化工具在底部显示了将“Who is Tom Cruise”查询转换为 SPARQL 查询的结果。你可以通过该网站尝试其他查询的效果。

## 1.2.9　机器阅读理解

**机器阅读理解**（machine comprehension，MC）能够根据段落回答问题，类似于学生们进行的阅读理解测试。图 1-9 是来自 AllenNLP 的可视化效果，它回答了“谁出演了《黑客帝国》”这一问题。图中显示了答案，以及整个段落。

Answer

Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano

Passage Context

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring **Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano**. It depicts a dystopian future in which reality as perceived by most humans is actually a simulated reality called "the Matrix", created by sentient machines to subdue the human population, while their bodies' heat and electrical activity are used as an energy source. Computer programmer "Neo" learns this truth and is drawn into a rebellion against the machines, which involves other people who have been freed from the "dream world."

图 1-9

通过在可视化图像中高亮特定单词，可以看出模型是如何运作的，如图 1-10 所示。

内部模型（测试版）

从段落到问题的注意力机制

对于段落里的每个单词，模型都会计算它对于被问到的单词的注意力。以下热图显示了注意力的大小情况并针对矩阵中的每一行进行了标准化。

Keanu
Reeves
,
Laurence
Fishburne
,
Carrie
-
Anne
Moss
,
Hugo
Weaving
,
and
Joe
Pantoliano

图 1-10

## 1.2.10　文字蕴含

**文本蕴含**（textual enrichment，TE）可以预测不同文本中的事实是否相同。图 1-11 对此进行了可视化描述。

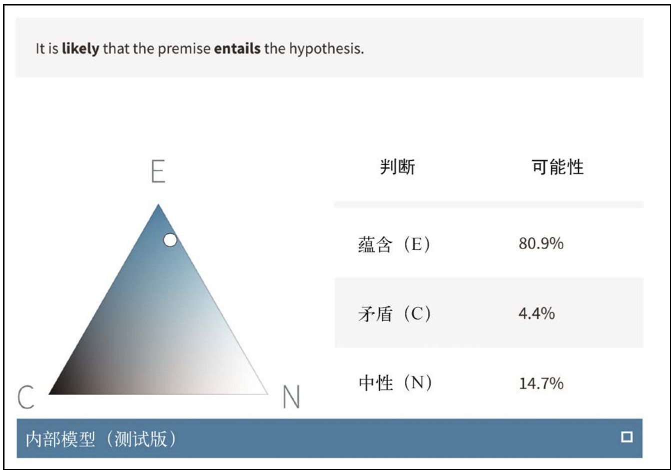



图　1-11

示例中的前提是“如果你帮助有需要的人，上天会奖赏你”，假设是“向穷人捐钱会产生良好的后果”。图 1-11 对蕴含、矛盾和中性结果的概率都进行了展示。

## 1.2.11　指代消解

当有多人进行互动时，代词解析会解决文本中代词的指代问题。图 1-12 对指代消解示例进行了可视化。

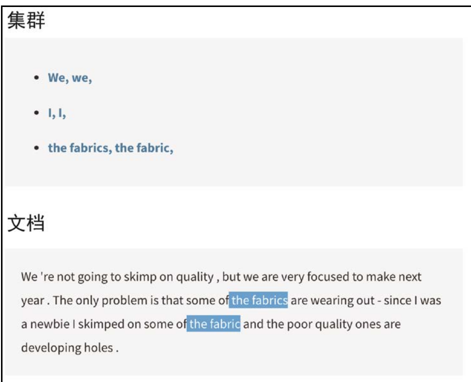



图　1-12

## 1.2.12　搜索

在网站上搜索信息是网络生活不可或缺的一部分，同时也是 NLP 技术的一种应用。本例中的搜索服务由 Bing API 提供，如图 1-13 所示。

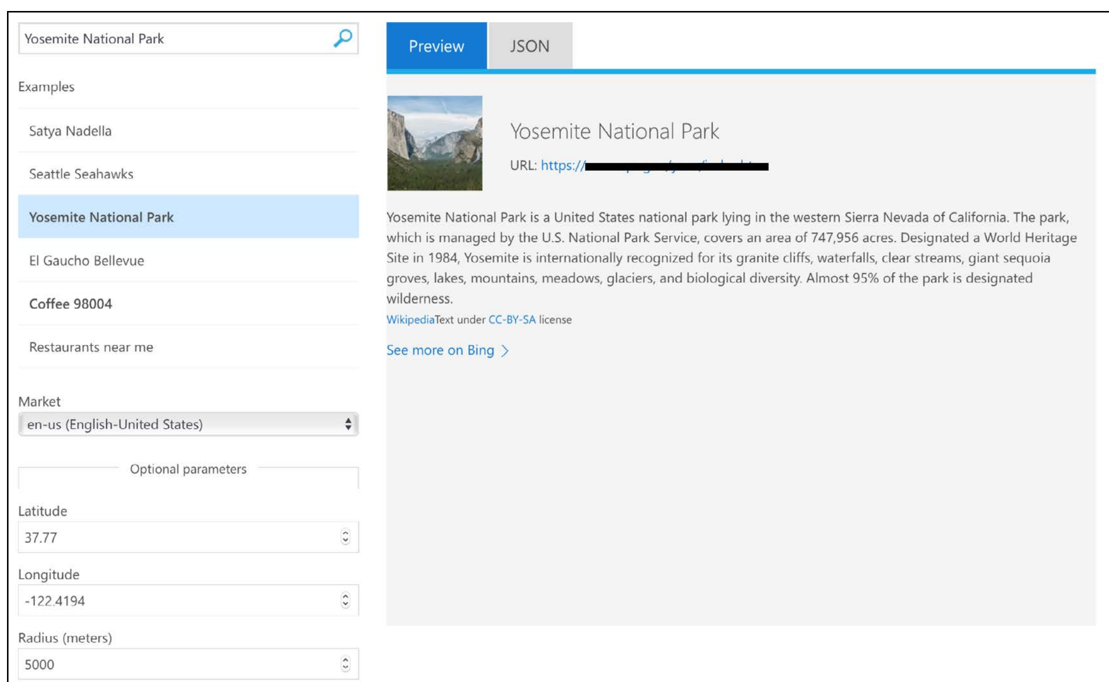



图 1-13　约塞米蒂国家公园的搜索结果

搜索 API 可以同应用集成在一起以使用户获得更好的体验。

## 1.2.13　问答和聊天机器人

当提供上下文和问题时，问答系统能够生成答案。图 1-14 展示了聊天机器人的范式。

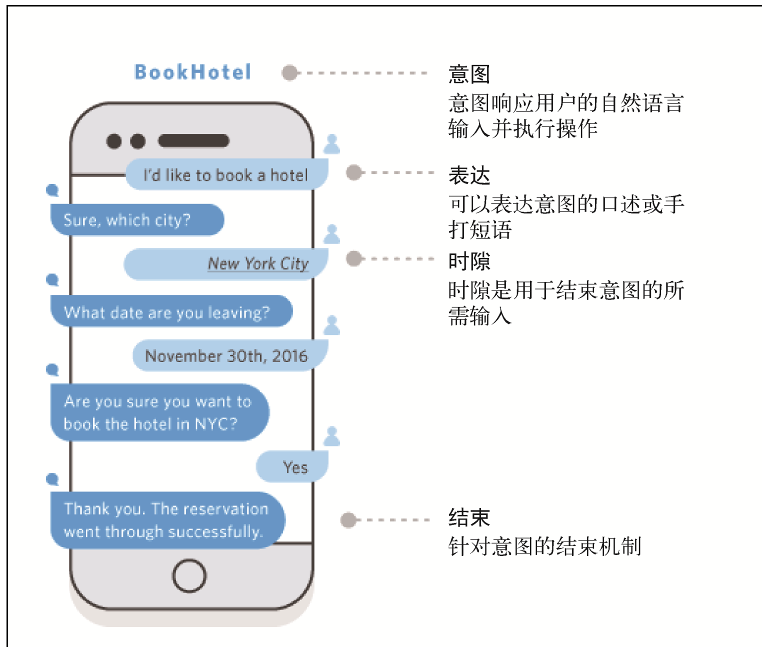



图　1-14

不同的应用会拥有其特定的聊天机器人。

### 1.2.14　文本转语音

有时候需要将文本转换为语音。对于个人机器人来说，能够与用户对话大有益处。

下面来看看如何利用 AWS API 实现 Amazon Polly 的文本转语音工程。使用该 API，我们可以输入文本并将其转换为语音。所生成的音频文件可以流式传输或下载。

转换的声音听起来应该足够自然以使用户产生共鸣。Google 以 30 种不同的声音、12 种不同的语言来提供文本转语音服务，且其语速和音高都是可调的。图 1-15 显示了一个文本转语音的示例，其中所有的参数均可调整。

Text to speak:

Google Cloud Text-to-Speech enables developers to synthesize natural-sounding speech with 32 voices, available in multiple languages and variants. It applies DeepMind's groundbreaking research in WaveNet and Google's powerful neural networks to deliver the highest fidelity possible. As an easy-to-use API, you can create lifelike interactions with your users, across many applications and devices.

text ssml

Language / locale: English (United States)　Voice type: WaveNet　Voice name: en-US-Wavenet-D　Speed: 1.00　Pitch: 0.00

Request URL

https://texttospeech.googleapis

Request body


```
{
  "audioConfig": {
    "audioEncoding": "LINEAR16",
    "pitch": "0.00",
    "speakingRate": "1.00"
  },
  "input": {
    "text": "Google Cloud Text-to-Speech enables developers to synthesize natural-sounding speech with 32 voices, available in multiple languages and variants. It applies DeepMind's groundbreaking research in WaveNet and Google's powerful neural networks to deliver the highest fidelity possible. As an easy-to-use API, you can create lifelike interactions with your users, across many applications and devices."
  },
  "voice": {
    "languageCode": "en-US",
    "name": "en-US-Wavenet-D"
  }
}
```


Hide JSON　SPEAK IT

图 1-15　参数微调

文本转语音的请求可以来自于任何已连接的设备（例如移动设备、汽车和电视机等）并用于客户服务、演示教育文本或动画内容。

### 1.2.15　语音转文本

有时也需要将语音转换为文本，也就是语音识别问题。Google 语音识别系统支持 120 种语言的转换。音频可以流式输入，也可以发送预先录好的视频。系统可以对不同的类别进行格式化，例如专有名词和标点符号。图 1-16 所示示例来自于 Google。

Language
English (United States)
Punctuation
Input type
Microphone　File upload

Request URL

https://speech.googleapis

Request body

```
{
  "audio": {
    "content": "/* Your audio */"
  },
  "config": {
    "enableAutomaticPunctuation": true,
    "encoding": "LINEAR16",
    "languageCode": "en-US",
    "model": "default"
  }
}
```

Hide JSON　START NOW

图　1-16

语音识别系统提供了用于视频、电话和搜索音频的不同模型。即使存在背景噪声，该功能也可以正常使用并过滤不当内容。

## 1.2.16　说话人识别

说话人识别是查找讲话人姓名的任务。 通过音频片段，系统可以识别出多个人的声音。

## 1.2.17　口语对话系统

口语对话系统的示例包括如 Google Voice、Apple Siri 和 Amazon Alex 等家庭助理。聊天机器人、语音转文本、文本转语音、说话人识别和搜索之类的所有应用都可以被组合起来，组成口语对话系统的相关体验。

## 1.2.18　其他应用

NLP 技术还有其他一些应用，以下列出了其中一部分。

- **垃圾邮件检测**：可以将我们收到的电子邮件分类为垃圾邮件或非垃圾邮件。
- **新闻分类**：基于数个类别对新闻项进行分类可能会很有帮助。
- **识别作者、性别或年龄**：可以从一段文本中检测出作者的性别和年龄。同样也可以用语音数据来标记相似的属性。
- **主题发现**：可以确定文章的主题。
- **文本生成**：机器生成的文本有很多有趣的应用。
- **语音翻译**：Skype 已启动了实时翻译功能，涉及语音转文本、机器翻译和文本转语音等技术。
- **文本摘要**：摘要任务将文本作为输入并输出文本的摘要。摘要通常比原始文本要短得多。例如在会议后，抄录的文本可以被摘要并发送给所有人。
- **段落理解**：段落理解是针对给定文章片段回答问题的高中级任务。
- **成分句法分析**：成分句法分析通过预测将句子的树型组成分为各个成分 。

## 1.3 小结

在本章中，你学习了 NLP 的基础知识，了解了一些可能会用到 NLP 的应用，也看到了云供应商提供的用于访问 NLP 应用的一些 API。在看完这些云产品后，你将在以后的章节中学习到各个应用背后的科学原理。

在下一章中，我们将介绍 NLTK 库的基础知识，覆盖基本的工程功能，并编写一个简单的文本分类任务。

第2章

# 使用 NLTK 进行文本分类和词性标注

**自然语言工具包**（Natural Language Toolkit，NLTK）是一个用于 NLP 任务的 Python 库，其功能涉及分词、分句、执行进阶任务（如语法分析和文本分类），等等。NLTK 提供了一些针对自然语言的模块和接口，可用于执行诸如文档主题识别、词性标注、情感分析等任务。为了实验各种 NLP 任务，NLTK 还提供了各种文本语料库的模块，从基本的文本集合到带标签的结构化文本（如 WordNet）。尽管 NLTK 库提供了大量 API，但我们仅介绍其在实际 NLP 应用中最为常见、最为重要的部分。

本章涵盖以下主题：

- 安装 NLTK 及其模块
- 文本预处理及探索性分析
- 词性标注
- 训练影评的情感分类器
- 训练词袋分类器

## 2.1 安装 NLTK 及其模块

在开始示例之前，我们将利用 NLTK 库及其 Python 依赖库准备好系统环境。Python 自带的 `pip` 安装工具可用来安装 NLTK 并可选地安装 `numpy`，如下所示：

```
sudo pip install -U nltk
sudo pip install -U numpy
```

NLTK 语料库及众多模块可以通过 Python 交互 shell 或 Jupyter Notebook 使用通用的 NLTK 下载器安装，代码如下所示。

```
import nltk
nltk.download()
```

该命令会打开如图 2-1 所示的 NLTK 下载器，你可以在其中选择所需的包或集合。

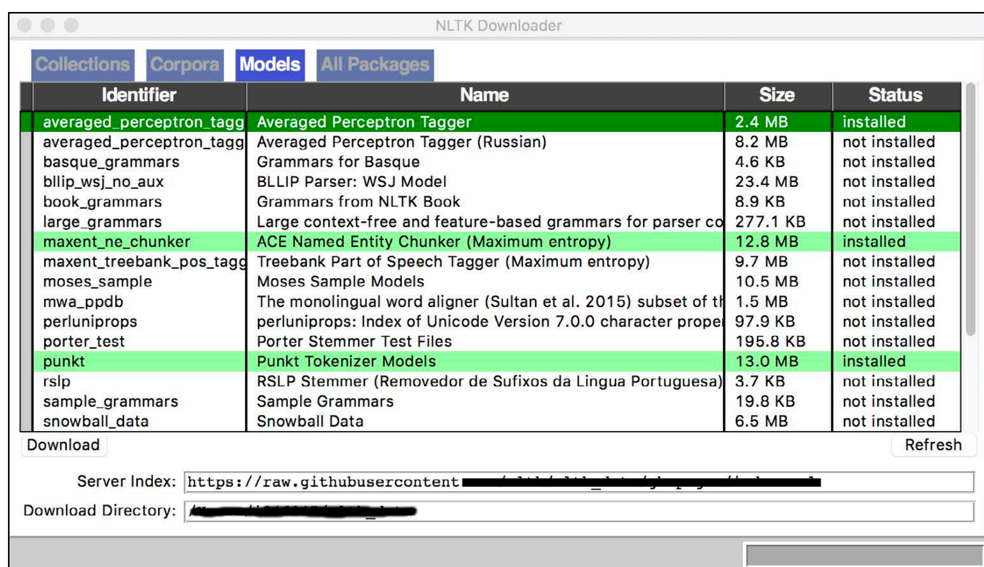



图　2-1

正如图 2-1 所示，特殊集合、文本语料库、NLTK 模型和包都可以被选中并安装。我们导航到 stopwords 并安装，以备之后使用。表 2-1 展示了本章样例所需安装的模块。

表　2-1

| No. | 包　　名 | 描　　述 |
|---|---|---|
| 1 | brown | Brown 文本语料库 |
| 2 | gutenberg | Gutenberg 文本语料库 |
| 3 | max_ne_chunker | 用于文本分块的模块 |
| 4 | movie_reviews | 电影评论情感极性数据 |
| 5 | product_reviews_1 | 基本电影评论语料库 |
| 6 | punkt | 分词、分句模块 |
| 7 | treebank | Peen Treebank 数据集样例 |
| 8 | twitter_samples | Twitter 消息样例 |
| 9 | universal_tagset | 通用词性标注映射 |
| 10 | webtext | Web 文本语料库 |
| 11 | wordnet | WordNet 语料库 |
| 12 | words | 单词列表 |

## 2.2　文本预处理及探索性分析

我们将首先通过一些基本的 NLP 实战任务（如文本预处理及探索性分析）来对于 NLTK 进行概述。文本预处理步骤涉及如**分词**、**词干提取**和**去除停用词**之类的任务。对准备好的文本数据进行探索性分析可以了解其主要特征，包括文本的主题和词频分布。

### 2.2.1　分词

单词词元（token）是任何 NLP 任务都会涉及的文本基本单元。处理文本时，第一步就是将文本拆分为词元。NLTK 为此提供了不同类型的**分词器**。我们将研究如何对来自 NLTK 中 Twitter 样例语料库的 Twitter 评论进行分词。从此刻开始，所有演示代码都可以通过在命令行上使用标准 Python 解释器来运行：

```
>>> import nltk
>>> from nltk.corpus import twitter_samples as ts
>>> ts.fileids()
['negative_tweets.json', 'positive_tweets.json', 'tweets.20150430-
223406.json']
>>> samples_tw = ts.strings('positive_tweets.json')
>>> samples_tw[100]
"@metalgear_jp @Kojima_Hideo I want you're T-shirts ! They are so cool ! :D"
>>> from nltk.tokenize import word_tokenize as wtoken
>>> wtoken(samples_tw[100])
['@', 'metalgear_jp', '@', 'Kojima_Hideo', 'I', 'want', 'you', "'re", 'T-shirts', '!',
'They', 'are', 'so', 'cool', '!', ':', 'D']
```

为实现基于标点和空格的文本分割，NLTK 也提供了能同时标注出标点符号的 `wordpunct_tokenize` 分词器。这一步骤可通过如下代码段说明：

```
>>> samples_tw[100]
"@metalgear_jp @Kojima_Hideo I want you're T-shirts ! They are so cool ! :D"
>>>from nltk.tokenize import wordpunct_tokenize
>>>wordpunct_tokenize(samples_tw[100])
['@', 'metalgear_jp', '@', 'Kojima_Hideo', 'I', 'want', 'you', "'", 're', 'T', '-',
'shirts', '!', 'They', 'are', 'so', 'cool', '!', ':', 'D']
```

正如你所见，与 `word_tokenize` 分词器不同的是，`wordpunct_tokenize` 分词器能将单词之间的连字符（-）和其他标点符号一样标注出来。我们也可以使用 NLTK 的正则表达式分词器实现自定义分词，如以下代码所示：

```
>>> from nltk import regexp_tokenize
>>> patn = '\w+'
>>> regexp_tokenize(samples_tw[100],patn)
['metalgear_jp', 'Kojima_Hideo', 'I', 'want', 'you', 're', 'T', 'shirts', 'They',
'are', 'so', 'cool', 'D']
```

在先前的代码中，我们使用了一个简单的正则表达式（`regexp`）去匹配只包含字母和数字字符的单词。在下面的另一个例子中，我们将使用正则表达式去匹配单词和部分标点符号。

```
>>> patn = '\w+|[!,\-,]'
>>> regexp_tokenize(samples_tw[100],patn)
['metalgear_jp', 'Kojima_Hideo', 'I', 'want', 'you', 're', 'T', '-', 'shirts', '!',
'They', 'are', 'so', 'cool', '!', 'D']
```

通过改变 regexp 的模式使其包含标点符号，我们就可以在结果中标注出这些标点符号。在 Python 的结果列表中，可以很明显地看到词元!和-都被展示了出来。



## 2.2.2 词干提取

词干提取是一种文本预处理任务，将单词的相关或相似变体（例如 walking）转换为其基本形式（例如 walk），因为它们具有相同的含义。词干提取转换的基本操作之一是将单词的复数形式还原为单数形式，例如将 apples 还原为 apple。尽管这是一个非常简单的转换，但确实存在更加复杂的操作。我们将使用 Martin Porter 提出的流行的 Porter 词干提取器来说明这一点，正如以下代码所示：

```
>>> import nltk
>>> from nltk.stem import PorterStemmer
>>> stemming = PorterStemmer()
>>> stemming.stem("enjoying")
'enjoy'
>>> stemming.stem("enjoys")
'enjoy'
>>> stemming.stem("enjoyable")
'enjoy'
```

本例中，词干提取将单词 enjoy 的不同动词形式（enjoying 和 enjoy）和形容词形式（enjoyable）简化为基本形式。词干提取器所使用的 Porter 算法利用各种特定语言的规则（本例中为英语）提取词干单词。如上述示例代码所示，一个规则是从单词中删除诸如 ing 之类的后缀。但词干提取所生成的并不总是单词，如下例所示：

```
>>> stemmer.stem("variation")
'variat'
>>> stemmer.stem("variate")
'variat'
```

在此，variat 并不是英文单词。对于法语、西班牙语和德语等其他语言，Nltk.stem.snowball 模块也提供了 snowball 词干提取器。它是一种词干提取语言，可用于在不同语言中创建词干提取的标准规则。我们可以使用以下正则表达式创建自定义的词干提取器，就像对分词器所做的那样：

```
>>> regexp_stemmer = RegexpStemmer("able$|ing$",min=4)
>>> regexp_stemmer.stem("flyable")
'fly'
>>> regexp_stemmer.stem("flying")
'fly'
```

正则模式 `able$|ing$` 去除了单词中可能出现的后缀 able 和 ing，而 `min` 则限定了提取词干单词的最小长度。

### 2.2.3 去除停用词

常用英文单词（例如 the、is 和 he 等）通常称为**停用词**。其他语言也有类似的常用词，同属这一类别。**去除停用词**是 NLP 应用中另一个常见的预处理步骤。在此步骤中，我们将删除那些对文档没有任何意义的词，例如语法冠词和代词。诸如 a、an、he 和 her 等单词都是需要去除的。停用词在整个文本中频繁出现，但它们本身可能不会对 NLP 任务（例如文本分类或搜索）产生任何影响。让我们通过以下代码看一下英文中停用词的示例：

```
>>> from nltk.corpus import stopwords
>>> sw_l = stopwords.words('english')
>>> sw_l[20:40]
['himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its',
'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
'who', 'whom', 'this']
```

上述代码只打印了前 20 个词项，所以输出只显示了英文中的部分停用词示例。在以下代码中，我们会研究这些词是如何被从文本中移除的：

```
>> example_text = "This is an example sentence to test stopwords"
>>> example_text_without_stopwords=[word for word in example_text.split()
if word not in sw_l]
>>> example_text_without_stopwords
['This', 'example', 'sentence', 'test', 'stopwords']
```

如你所见，有些词（如 an、is 和 to）已经被移除了。正如例子中那样，除英语外，NLTK 还为 21 种语言提供了停用词语料库。在以下示例中，我们使用代码来看看在特定文本语料库中停用词所占的比例：

```
>> from nltk.corpus import gutenberg
>>> words_in_hamlet = gutenberg.words('shakespeare-hamlet.txt')
>>> words_in_hamlet_without_sw = [word for word in words_in_hamlet if word
not in sw_l]
>>> len(words_in_hamlet_without_sw)*100.0/len(words_in_hamlet)
69.26124197002142
```

以上示例表明，在莎士比亚的《哈姆雷特》中，文本的很大一部分（约 30%）是由停用词组成的。在许多 NLP 任务中，停用词并不重要，所以可以在预处理过程中移除。

### 2.2.4 探索性分析

获得词元数据后，常用的基本分析之一是对单词或词元及其在文档中的分布进行计数，从而更多地了解文档中的主要话题。让我们从分析 NLTK 自带的网络文本数据开始吧：

```
>>> import nltk
>>> from nltk.corpus import webtext
>>> webtext_sentences = webtext.sents('firefox.txt')
>>> webtext_words = webtext.words('firefox.txt')
>>> len(webtext_sentences)
1142
>>> len(webtext_words)
102457
```

请注意，尽管网络文本数据也包含其他数据（例如广告和电影的台本），但我们仅加载与 Firefox 论坛相关的文本（firefox.txt）。先前的代码输出分别给出了句子和单词在整个文本语料库中的数量。我们还可以通过将词汇表传递到集合中来获得词汇表的大小，如以下代码所示：

```
>>> vocabulary = set(webtext_words)
>>> len(vocabulary)
8296
```

为了获得文本的频数分布，可以利用 `nltk.FreqDist()` 函数来获取文本中使用最为频繁的单词。这可以提供关于文本数据主旨的大致思路，如以下代码所示：

```
>>> frequency_dist = nltk.FreqDist(webtext_words)
>>> sorted(frequency_dist,key=frequency_dist.__getitem__,
reverse=True)[0:30]
['.', 'in', 'to', '"', 'the', "'", 'not', '-', 'when', 'on', 'a', 'is',
't', 'and', 'of', '(', 'page', 'for', 'with', ')', 'window', 'Firefox',
'does', 'from', 'open', ':', 'menu', 'should', 'bar', 'tab']
```

以上结果给出了文本中使用最为频繁的 30 个单词，其中一些停用词（例如 the）显然经常在英语中出现，但是仍可以看到诸如 Firefox 之类的词也出现了，这是因为我们用于分析的文本来自有关 Firefox 浏览器的论坛。我们还可以通过以下代码查看字母长度大于 3 的单词的频数分布情况，这将排除诸如 and 和 is 之类的单词：

```
>>> large_words = dict([(k,v) for k,v in frequency_dist.items() if
len(k)>3])
>>> frequency_dist = nltk.FreqDist(large_words)
>>> frequency_dist.plot(50,cumulative=False)
```

此处，我们筛选出了所有字母长度大于 3 的单词并创建了词频元组的字典。该字典将被传递到 NLTK 频数分布画图工具中。我们现在可以看到单词的频数分布图如图 2-2 所示。

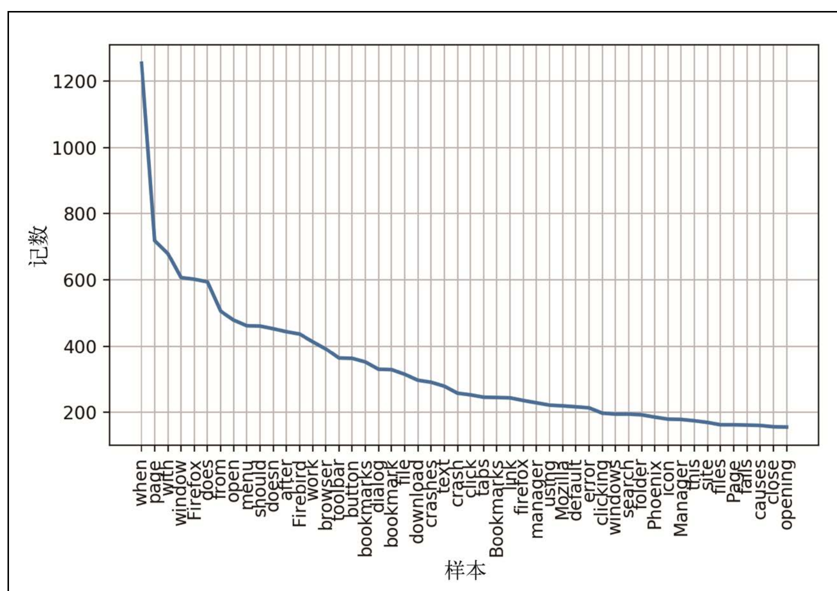



图　2-2

图 2-2 显示了排名前 50 的单词的频数分布。根据其分布情况，我们可以生成一个词云（word cloud）以直观、可视化地看到文本中单词的使用情况。为此，必须安装 Python 中的 `wordcloud` 包，安装代码如下所示：

```
pip install wordcloud
```

以上代码能够实现 `wordcloud` 包的安装。该包将单词随机地放置在幕布上，单词大小与其在文本中的频数成正比，以此生成词云。我们现在来看看展示词云的代码，如下所示：

```
>>> from wordcloud import WordCloud
>>> wcloud = WordCloud().generate_from_frequencies(frequency_dist)
>>> import matplotlib.pyplot as plt
>>> plt.imshow(wcloud, interpolation='bilinear')
>>> plt.axis("off")
(-0.5, 399.5, 199.5, -0.5)
>>> plt.show()
```

在以上代码中，我们输入了先前通过 NLTK 库获得的单词频数分布，对应生成的词云如图 2-3 所示。

图　2-3

基于先前有关停用词的示例，我们来看看移除停用词后词频分布是如何改变的。当移除停用词后，词云看上去与文本的主题更为契合，如图 2-4 所示。

图　2-4

在以上词云图中，when、with 和 from 等常用单词都被移除了。可以通过以下代码检查单词频数分布词典对此进行验证：

```
>>> words_in_webtext_without_sw = [word for word in webtext_words if word
not in sw_l]
>>> 'when' in words_in_webtext_without_sw
False
>>> 'from' in words_in_webtext_without_sw
False
```

同理，对于在去除停用词之前云图中出现的其他单词，我们还可以检查其在 `words_in_webtext_withour_sw` 频数分布字典中是否仍然存在。

# 2.3　词性标注

我们已经分析了一些基本的 NLP 预处理任务，例如分词、词干提取和移除停用词等，也探索了如何确定文本语料库中单词的分布并进行可视化。本节将深入探索 NLTK，了解其词性标注功能。

## 2.3.1　词性标注定义

词性标注将句子中的单词以不同语义功能或语法功能进行分类。在英语中，主要的词性为**名词**、**代词**、**形容词**、**动词**、**副词**、**介词**、**限定词**和**连词**，而词性标注正是为文本中的每个单词或词元附加这些类别之一。NLTK 提供了标注好的文本语料库和一组词性训练器，用以创建自定义的标注器。NLTK 中最常见的标注数据集是 Penn Treebank 和 Brown Corpus，前者由经过分析的期刊日志、电话交谈等文本集合组成，而后者则由 15 种不同类别（科学、政治、宗教和体育等）的文章组成。这些文本数据提供了细粒度标注，然而许多应用可能只需要以下的通用标注集。

- `VERB`：动词（所有时态和方式）
- `NOUN`：名词（普通名词、专有名词）
- `PRON`：代词
- `ADJ`：形容词
- `ADV`：副词
- `ADP`：介词（前置词、后置词）
- `CONJ`：连词
- `DET`：限定词
- `NUM`：基数
- `PRT`：小品词或其他功能词
- `X-other`：外来词、错别字、缩写
- `.`：标点符号

NLTK 还提供了从带标注语料库（例如 Brown Corpus）到通用标签的映射，如以下代码所示。与通用标签集相比，Brown Corpus 的词性标注粒度更细。例如，`VBD` 标注（用于过去式动词）和 `VB` 标注（用于基本形式动词）会被映射到通用标注集中的 `VERB`：

```
>>> from nltk.corpus import brown
>>> brown.tagged_words()[30:40]
[('term-end', 'NN'), ('presentments', 'NNS'), ('that', 'CS'), ('the',
'AT'), ('City', 'NN-TL'), ('Executive', 'JJ-TL'), ('Committee', 'NN-TL'),
(',', ','), ('which', 'WDT'), ('had', 'HVD')]
>>> brown.tagged_words(tagset='universal')[30:40]
[('term-end', 'NOUN'), ('presentments', 'NOUN'), ('that', 'ADP'), ('the',
'DET'), ('City', 'NOUN'), ('Executive', 'ADJ'), ('Committee', 'NOUN'),
(',', '.'), ('which', 'DET'), ('had', 'VERB')]
```

这里，你可以看到 City 一词被标注为 NP-TL，表示它是在 Brown Corpus 中标题（TL）上下文里出现的专有名词（NP）。它被映射到通用标注集中的 NOUN。有时一些 NLP 任务可能需要这些更精细粒度的类别而非常规的通用标注。



### 2.3.2 词性标注的应用

词性标注可在命名实体识别（NER）、情感分析、问答和单词消歧中得到应用。我们将在以下代码中查看单词歧义消除的示例。在句子“I left the room”和“Left of the room”中，left 一词表达了不同的含义，而词性标注器将有助于区分其意义。现在我们将研究如何标注出同一单词的两种不同用法：

```
>>> import nltk
>>> text1 = nltk.word_tokenize("I left the room")
>>> text2 = nltk.word_tokenize("Left of the room")
>>> nltk.pos_tag(text1,tagset='universal')
[('I', 'PRON'), ('left', 'VERB'), ('the', 'DET'), ('room', 'NOUN')]
>>> nltk.pos_tag(text2, tagset='universal')
[('Left', 'NOUN'), ('of', 'ADP'), ('the', 'DET'), ('room', 'NOUN')]
```

单词 left 在第一个示例中是动词，而在第二个示例中则是名词。在 NER 中，词性标注有助于基于标签完成人员、地点或位置的识别。NLTK 内置了一个训练好的分类器用以识别文本中的实体。该分类器可以在经过词性标注的句子上运行，如以下代码所示：

```
>>> import nltk
>>> example_sent = nltk.word_tokenize("The company is located in South
Africa")
>>> example_sent
['The', 'company', 'is', 'located', 'in', 'South', 'Africa']
>>> tagged_sent = nltk.pos_tag(example_sent)
>>> tagged_sent
[('The', 'DT'), ('company', 'NN'), ('is', 'VBZ'), ('located', 'VBN'),
('in', 'IN'), ('South', 'NNP'), ('Africa', 'NNP')]
>>> nltk.ne_chunk(tagged_sent)
Tree('S', [('The', 'DT'), ('company', 'NN'), ('is', 'VBZ'), ('located',
'VBN'), ('in', 'IN'), Tree('GPE', [('South', 'NNP'), ('Africa', 'NNP')])])
```

在示例语句中，ne_chunk() 函数使用训练好的**命名实体划分器**将 South Africa 标注为地缘政治实体（GPE）。到目前为止，我们已经看到了使用 NLTK 内置标注器的示例，而在下一节中，我们将研究如何开发自定义词性标注器。

### 2.3.3 训练词性标注器

现在，我们将使用 NLTK 的标注集语料库和 sklearn 随机森林机器学习模型来训练自己的词性标注器。本节的完整 Jupyter Notebook 文件可在本书代码库中找到（Chapter02/01_example.ipynb）。整个过程将是一个分类任务，因为我们需要预测句子中给定单词的词性标签。我们将使

用带有词性标注的 NLTK treebank 数据集作为训练数据或标注数据，并提取单词的前缀、后缀以及文本中的前序单词和相邻单词作为训练的特征。这些 特征对于将单词划分为不同词性具有良好的指示效果。以下代码显示了特征的提取过程：

```
def sentence_features(st, ix):
    d_ft = {}
    d_ft['word'] = st[ix]
    d_ft['dist_from_first'] = ix - 0
    d_ft['dist_from_last'] = len(st) - ix
    d_ft['capitalized'] = st[ix][0].upper() == st[ix][0]
    d_ft['prefix1'] = st[ix][0]
    d_ft['prefix2'] = st[ix][:2]
    d_ft['prefix3'] = st[ix][:3]
    d_ft['suffix1'] = st[ix][-1]
    d_ft['suffix2'] = st[ix][-2:]
    d_ft['suffix3'] = st[ix][-3:]
    d_ft['prev_word'] = '' if ix==0 else st[ix-1]
    d_ft['next_word'] = '' if ix==(len(st)-1) else st[ix+1]
    d_ft['numeric'] = st[ix].isdigit()
    return d_ft
```

函数 `statement_features()`将文本输入转换为特征字典 `d_ft`。每个句子以 Python 列表的形式与当前单词的索引一起被传入，以提取该单词的特征，其中索引 `ix` 用于获取相邻单词的特征以及单词的前缀/后缀。在后续实例中，我们将在训练后查看这些特征的重要性。现在我们将使用上一小节所说带有通用标注的 `treebank` 标注句子作为标注数据或训练数据：

```
tagged_sentences = nltk.corpus.treebank.tagged_sents(tagset='universal')
```

为简便起见，我们使用了通用标注，如传递给 `tags_sents` 函数中名为 `tagset` 的参数。除了通用标注外，还可以使用细粒度的 `treebank` 词性标注，而这将导致大量标签的出现。现在我们将为语料库中的每个标注语句提取特征并进行训练，所提取的特征存储在变量 `x` 中，而词性标注或者标签存储在变量 `y` 中。特征提取过程如以下代码所示：

```
def ext_ft(tg_sent):
    sent, tag = [], []

    for tg in tg_sent:
        for index in range(len(tg)):
            sent.append(sentence_features(get_untagged_sentence(tg),
index))
            tag.append(tg[index][1])

    return sent, tag

X,y = ext_ft(tagged_sentences)
```

在 sklearn 库中，我们利用 `DictVectorizer` 将特征值字典转换为训练向量或实例。应当注意的是，对于字符串类型的值，`DictVectorizer` 将其转换为独热编码向量。如果 `suffix3` 的可能特征取值的数量为 50，则输出中将有 50 个特征。我们将使用以下代码来应用 `DictVectorizer`：

```
n_sample = 50000
dict_vectorizer = DictVectorizer(sparse=False)
X_transformed = dict_vectorizer.fit_transform(X[0:n_sample])
y_sampled = y[0:n_sample]
```



本例使用了大约 50 000 个句子的样本量以加快训练速度，这些训练实例进一步分为 80%的训练集和 20%测试集（请参阅 Notebook）。本例使用 sklearn 的 RandomForestClassifier 集成分类器作为词性标注模型，如以下代码所示：

```
rf = RandomForestClassifier(n_jobs=4)
rf.fit(X_train,y_train)
```

经过训练后，我们可以使用示例句子来验证词性标注器的效果。在将例句传给 predict() 函数之前，我们将使用用于 NLTK 标注数据的函数（sentence_features()）来提取特征，如以下代码所示：

```
def predict_pos_tags(sentence):
 tagged_sentence = []
 features = [sentence_features(sentence, index) for index in
range(len(sentence))]
 features = dict_vectorizer.transform(features)
 tags = rf.predict(features)
 return zip(sentence, tags)
```

使用 sentence_features()函数将 sentence 列表变量中的单词转换为相应的特征，并使用先前训练好的 dict_vectorizer 对从函数提取的特征字典进行向量化处理：

```
test_sentence = "This is a simple POS tagger"
for tagged in predict_pos_tags(test_sentence.split()):
  print(tagged)
```

我们将测试语句作为单词列表，传递给 predict_pos_tags 函数，输出句子中每个单词的词性。以下输出显示了例句词性标注结果：

```
('This', 'DET')
('is', 'VERB')
('a', 'DET')
('simple', 'ADJ')
('POS', 'NOUN')
('tagger', 'NOUN')
```

例句的输出看起来很合理：它可以识别句子中的限定词、动词、形容词和名词。为了评估标注器的准确率，我们可以使用以下代码来预测测试数据的词性标注：

```
predictions = rf.predict(X_test)
accuracy_score(y_test,predictions)

Output
0.94520000000000004
```

预测的准确率大约为 94%，这看起来是合理的。我们还将查看混淆矩阵以观察标注器对于每个词性标签的性能，这将利用 sklearn 中的 `confusion_matrix` 函数，代码如下所示：

```
conf_matrix = confusion_matrix(y_test,predictions)
plt.figure(figsize=(10,10))
plt.xticks(np.arange(len(rf.classes_)),rf.classes_)
plt.yticks(np.arange(len(rf.classes_)),rf.classes_)
plt.imshow(conf_matrix,cmap=plt.cm.Blues)
plt.colorbar()
```

在用于绘制混淆矩阵的代码中，我们使用了来自随机森林分类器中的类作为 x 标签和 y 标签。这些标签是训练数据的词性标注。图 2-5 是混淆矩阵的图像表示。

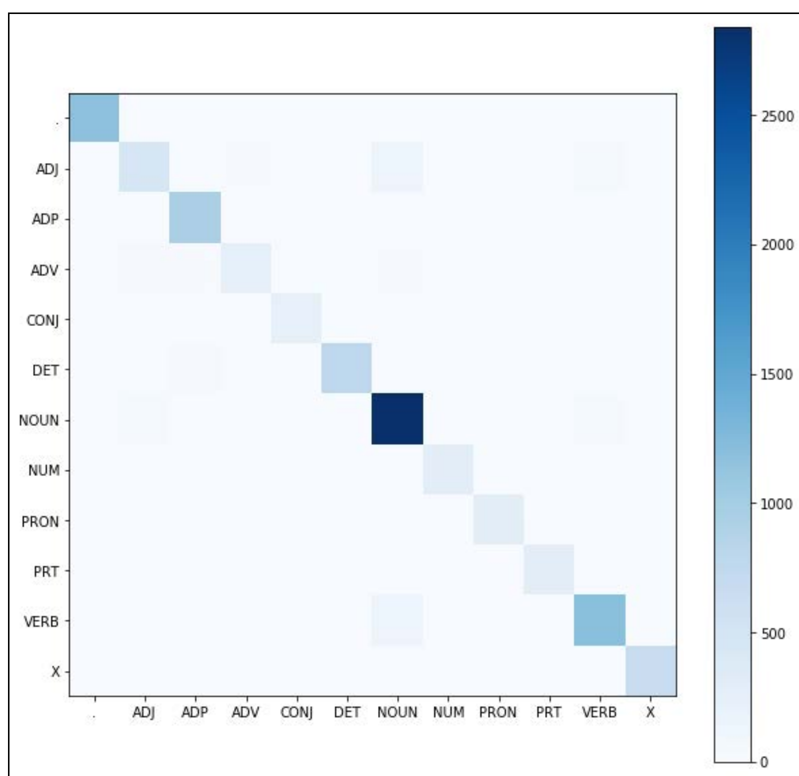



图　2-5

结果表明标注器对于句子中名词、动词和限定词方面的标注表现相对较好，这在图中深色部分得到了体现。现在，我们将通过以下代码查看模型的主要特征：

```
feature_list =
zip(dict_vectorizer.get_feature_names(),rf.feature_importances_)
sorted_features = sorted(feature_list,key=lambda x: x[1], reverse=True)
print(sorted_features[0:20])
```

随机森林模型中特征的权重存储在 Python 的 `feature_importances` 列表中。我们将按照特征权重对该列表进行降序排列，并使用先前的代码打印前 20 个特征，结果如下所示：

```
Output:

[('next_word=', 0.020920214730751722), ('capitalized',
0.01772036411509819), ('prefix1=,', 0.017100349286406635), ('suffix1=,',
0.013300188138108692), ('suffix2=ed', 0.012324641839199037), ('prefix1=*',
0.01184006667636649), ('suffix2=he', 0.010212280707210959), ('prefix2=th',
0.01012750927310713), ('suffix2=to', 0.010110760622078928), ('prefix3=the',
0.0094462675592230805), ('dist_from_first', 0.0093968467476374141),
('suffix1=f', 0.0092678798994399649), ('word=the', 0.0091584437614083847),
('dist_from_last', 0.0087969654754903419), ('prefix2=to',
0.0086095477647111125), ('suffix1=d', 0.0082316431932524976), ('word=a',
0.0077318882551946199), ('prefix2=an', 0.0074132280379715434),
('suffix1=s', 0.0067561700034315057), ('word=and', 0.0065749584774608179)]
```

可以看出某些后缀特征获得了更高的权重得分，例如以 ed 结尾的单词通常是过去式的动词，还可以发现一些标点符号（例如逗号）也会影响标注。尽管词性标注也是一种文本分类，但接下来我们将继续研究另一种常见的 NLP 任务——情感分类。

## 2.4 训练影评情感分类器

现在我们将对 NLTK 中的影评语料库进行情感分类。该示例的完整 Jupyter Notebook 文件可以在本书代码库中找到（Chapter02/02_example.ipynb）。

首先我们将根据情感类别（正面或负面）使用以下代码加载电影评论：

```
cats = movie_reviews.categories()
reviews = []
for cat in cats:
    for fid in movie_reviews.fileids(cat):
        review = (list(movie_reviews.words(fid)),cat)
        reviews.append(review)
random.shuffle(reviews)
```

`category()` 函数将返回 `pos` 或 `neg`，分别代表正面或负面情绪。这两个类别中分别有 1000 条评论，我们将使用 Python 中的 `random.shuffle()` 函数将它们由分组顺序混洗为随机顺序。接下来使用以下代码选择评论中的重点词，并将其用作特征工程或提取过程的基础词汇：

```
all_wd_in_reviews = nltk.FreqDist(wd.lower() for wd in
movie_reviews.words())
top_wd_in_reviews = [list(wds) for wds in
zip(*all_wd_in_reviews.most_common(2000))][0]
```

我们在影评中选择了出现最为频繁的前 2000 个单词进行特征生成。根据评论中是否存在这些词生成二进制特征，因此每个训练集将具有 2000 个特征：当单词在影评中出现时，对应特征

设为 1，否则设为 0。

```
def ext_ft(review,top_words):
    review_wds = set(review)
    ft = {}
    for wd in top_words:
        ft['word_present({})'.format(wd)] = (wd in review_wds)
    return ft
```

每条电影评论都将被传递给 ext_ft()函数并由该函数返回包含二进制特征的字典，具体代码如下所示：

```
featuresets = [(ext_ft(d,top_wd_in_reviews), c) for (d,c) in reviews]
train_set, test_set = featuresets[200:], featuresets[:200]
```

我们将标注好的数据按照 80%和 20%的比例分为训练集和测试集。作为初始测试，我们将使用 NLTK 自带的简单的朴素贝叶斯分类器（naïve Bayes classifier），如以下代码所示：

```
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))


Output
0.805
```

即使使用简单的朴素贝叶斯分类器，仍可以达到 80%的准确率。同时，我们还可以看到分类模型所学习到的信息量最为丰富的特征。这里使用以下代码展示对应的前 20 个特征：

```
classifier.show_most_informative_features(10)
```

show_most_informative_features 函数以输出优先特征的数量为参数并输出相关特征，结果如下所示：

```
Output:

Most Informative Features
word_present(seagal) = True            neg : pos      =      12.9 : 1.0
word_present(outstanding) = True       pos : neg      =      10.2 : 1.0
word_present(mulan) = True             pos : neg      =       7.0 : 1.0
word_present(wonderfully) = True       pos : neg      =       6.5 : 1.0
word_present(damon) = True             pos : neg      =       5.7 : 1.0
word_present(ridiculous) = True        neg : pos      =       5.6 : 1.0
word_present(awful) = True             neg : pos      =       5.6 : 1.0
word_present(lame) = True              neg : pos      =       5.5 : 1.0
word_present(era) = True               pos : neg      =       5.4 : 1.0
word_present(waste) = True             neg : pos      =       5.3 : 1.0
```

模型学到的 waste、awful 和 ridiculous 等词语看起来传达了负面的含义，而 outstanding、wonderfully 和 era 等词语传达了积极的情感。

现在，我们将使用 sklearn 中的随机森林分类器模型及影评数据进行评估。但是在此之前，

要使用 DictVectorizer 对特征进行向量化，就像我们在上一节中训练词性标注器所做的那样，如以下代码所示：

```
d_vect=None
def get_train_test(tr_set,te_set):
    global d_vect
    d_vect = DictVectorizer(sparse=False)
    X_tr, y_tr = zip(*tr_set)
    X_tr = d_vect.fit_transform(X_tr)
    X_te,y_te = zip(*te_set)
    X_te = d_vect.transform(X_te)
    return X_tr,X_te,y_tr,y_te
```

tr_set 和 te_set 是我们先前获得的训练集和测试集实例，而 get_train_test 函数则返回用于传递给 sklearn 随机森林分类器的向量化特征，如以下代码所示：

```
X_train,X_test,y_train,y_test = get_train_test(train_set,test_set)
rf = RandomForestClassifier(n_estimators=100,n_jobs=4,random_state=10)
rf.fit(X_train,y_train)
```

在这里使用了 100 个分类器（或者说决策树）用于整体分类器，其中参数 n_jobs 表示并行工作的数量，通过并行可以加速训练和预测的过程。预测评估代码及结果如以下代码所示：

```
preds = rf.predict(X_test)
print(accuracy_score(y_test,preds))

Output
0.81
```

与朴素贝叶斯分类器相比，随机森林的准确率略有提高，大约为 81%。现在我们将删除评论中的所有停用词，并再次训练分类器以观察模型准确率是否有所提升。我们利用 NLTK 停用词语料库来删除停用词，并像之前一样选择前 2000 个最为常见的单词，如以下代码所示：

```
from nltk.corpus import stopwords
stopwords_list = stopwords.words('english')
all_words_in_reviews = nltk.FreqDist(word.lower() for word in
movie_reviews.words() if word not in stopwords_list)
top_words_in_reviews = [list(words) for words in
zip(*all_words_in_reviews.most_common(2000))][0]
```

此时 top_words_in_reviews 已经移除了停用词，我们将以此作为词汇生成特征并训练随机森林分类器：

```
preds = rf.predict(X_test)
print(accuracy_score(y_test,preds))
0.76
```

实际上，删除该数据集的停用词并没有提高模型的表现，反而使准确率有所降低。通过使用以下代码，可以发现信息量最大的特征，就像对贝叶斯分类器所做的那样：

```
features_list =
zip(dict_vectorizer.get_feature_names(),rf.feature_importances_)
features_list = sorted(features_list, key=lambda x: x[1], reverse=True)
print(features_list[0:20])
```

像之前一样，我们对随机森林分类器所学习到的特征进行基于权重的排序，并且从排序后的 Python 列表中打印前 20 个：

```
[('word_present(bad)', 0.012904816953952729), ('word_present(boring)',
0.006797056379259946), ('word_present(stupid)', 0.006742453545126172),
('word_present(awful)', 0.006057321244427093), ('word_present(worst)',
0.005618499631730539), ('word_present(waste)', 0.005091242651240423),
('word_present(supposed)', 0.005019844359438753),
('word_present(excellent)', 0.005002846831984908), ('word_present(mess)',
0.004735341799753426), ('word_present(wasted)', 0.004477280752464545),
('word_present(ridiculous)', 0.00435578373608493), ('word_present(lame)',
0.00404257877140679), ('word_present(also)', 0.003663095965733155),
('word_present(others)', 0.0035194019538410553), ('word_present(dull)',
0.003464806019875671), ('word_present(plot)', 0.0034406946286116035),
('word_present(nothing)', 0.0033285487918061265),
('word_present(performances)', 0.003286015291474251),
('word_present(outstanding)', 0.0032708132090801516),
('word_present(memorable)', 0.003265718932501386)]
```

类似于朴素贝叶斯分类器，我们也可以找到传达正面和负面情绪的词语。虽然二进制特征可能对基本文本分类任务很有用，但是不适用于更为复杂的文本分类应用。因此下一节将介绍更为先进的特征提取技术。

## 2.5　训练词袋分类器

在上一节中，我们对影评中的单词提取了简单的二进制特征用以了解正面和负面的情感。相较于该方法，更好的做法是使用隐含特征，如文本中单词的使用频率。与表示单词存在与否的二进制特征相比，单词计数可以更好地捕获文本或文档的特征。**词袋**作为文本的向量表示，其中每个向量维都能捕获文本中单词的出现频率、存在与否或加权值，但是不能捕获单词之间的顺序。

因此上一节中讨论的二进制特征提取是文本的简单词袋表示，而现在我们要使用一个更好的词袋表示法对 Twitter 文本进行情感分类。该示例的完整 Jupyter Notebook 可在本书代码库中找到（Chapter02/03_example.ipynb）。本例将使用 NLTK 中的 Twitter 样本语料库，正如 `movie_reviews` 语料库一样，Twiiter 样本语料库也包含了情感的极性。

```
pos_tweets = [(string, 1) for string in
twitter_samples.strings('positive_tweets.json')]
neg_tweets = [(string,0) for string in
twitter_samples.strings('negative_tweets.json')]
pos_tweets.extend(neg_tweets)
comb_tweets = pos_tweets
random.shuffle(comb_tweets)
tweets,labels = (zip(*comb_tweets))
```

像以前一样，我们从 JSON 文件中读取数据并附加情感标签。JSON 解析和文本提取过程由 NLTK 使用字符串功能完成，而我们在情感标签上附上 1 表示正面情绪，附上 0 表示负面情绪。同时，我们还会在推文的 Python 列表及情感标签元组中进行混洗以调整正面和负面情绪的顺序，如以下代码所示：

```
count_vectorizer = CountVectorizer(ngram_range=
  (1,2),max_features=10000)
X = count_vectorizer.fit_transform(tweets)
```

我们利用 sklearn 中的 `CountVectorizer` 函数生成特征并将特征数量限制为 10 000。我们还使用了一元（unigram）和二元（bigram）特征。一个 *n* 元表示从文本中连续采样的 *n* 个单词特征。一元模型是通常的单个单词特征，而二元模型则是文本中两个连续的单词序列。因为二元模型是两个连续的单词，所以可以捕获文本中的短单词序列或者短语。在此示例中，由于 `ngram_range` 为`(1,2)`，`CountVectorizer` 将既从推文中提取一元特征又提取二元特征。

在将其分为 80%的训练集及 20%的测试集后，我们将使用推文训练模型，如以下代码所示：

```
rf = RandomForestClassifier(n_estimators=100,n_jobs=4,random_state=10)
rf.fit(X_train,y_train)
X_train,X_test,y_train,y_test =
train_test_split(X,labels,test_size=0.2,random_state=10)
```

现在我们将通过测试集预测其情感标签来对模型进行评估，得出准确率得分和混淆矩阵：

```
preds = rf.predict(X_test)
print(accuracy_score(y_test,preds))
print(confusion_matrix(y_test,preds))

Output

0.758
[[796 173]
[311 720]]
```

该模型的准确率约为 75%。接下来我们将使用 `tfidf` 向量化器对模型展开测试，该向量化器似于基于计数的 *n* 元语法模型，不同之处在于它对计数进行加权：根据单词在所有文档或文本中的出现情况为单词赋予权重。这意味着相较于只在特定文档中出现的单词，在所有文档里都频繁出现的单词的权重会更低。

```
from nltk.corpus import stopwords
tfidf = TfidfVectorizer(ngram_range=(1,2),max_features=10000)
X = tfidf.fit_transform(tweets)
```

如先前所做的那样，我们从文本中提取一元模型和二元模型，并使用测试数据对模型展开评估：

```
preds = rf.predict(X_test)
print(accuracy_score(y_test,preds))
```

```
print(confusion_matrix(y_test,preds))
```

```
Output

0.756
```

此时 `TfidfVectorizer` 仍未提高模型的准确率。我们将使用 NLTK 停用词集从推文中去除停用词：

```
from nltk.corpus import stopwords
tfidf = TfidfVectorizer(ngram_range=(1,2),max_features=10000,
stop_words=stopwords.words('english'))
X = tfidf.fit_transform(tweets)

preds = rf.predict(X_test)
print(accuracy_score(y_test,preds))
print(confusion_matrix(y_test,preds))
```

```
Output

0.736
```

对测试数据的评估显示：模型的准确率有所下降。去除停用词可能并不总会提高准确率，因为准确性还取决于训练数据。特定的停用词可能会出现在指示推文情感的常见短语中。

## 2.6　小结

本章介绍了常见的 NLP 任务，例如使用 NLTK 库对文本进行预处理和探索性分析。真实世界中数据的非结构化特性需要进行大量预处理，例如分词、词干提取和停用词去除等，以使数据适应机器学习的需求。正如示例所示，NLTK 提供了广泛的 API 来执行这些预处理步骤：它提供了内置的包和模块，并支持灵活地构建自定义模块，例如用户自定义词干分析器和分词器。

我们还讨论了使用 NLTK 进行词性标注。词性标注是另一种常见的 NLP 任务，用于解决诸如单词消歧和问答之类的问题。情感分类等应用由于其研究价值和商业价值而被广泛使用。讨论情感分析时，我们使用了 NLTK 语料库和 sklearn 库，介绍了用于推文和影评文本分类的基本示例。以上示例可在简单的 NLP 任务中使用，而我们在后续章节中将介绍如何使用深度学习进行更为复杂的文本分类任务。

第 3 章

# 深度学习和 TensorFlow

由于采用了深度学习模型，利用 NLP 技术的应用已开始在语言翻译、文本摘要和文本转语音等任务中达到接近人类水平的准确性。推动这一进展的是深度学习领域的两个关键发展。第一个是全新深度神经网络结构探索工作的迅速进展，而这是通过海量数据的获取实现的。与传统方法相比，此类架构可以实现卓越的性能表现。第二个则是开源工具或库（例如 TensorFlow）的可用性不断提高，使得这些现代架构可以在实际或生产应用中轻松实现。本章的目的是提供深度学习和 TensorFlow 的必要基础知识，使你能够充满信心地阅读之后的章节。

本章涵盖以下主题：

- 深度学习中的各种概念和术语
- 通用深度学习架构（如 CNN 和 RNN）的概述
- TensorFlow 的安装、设置及入门

## 3.1 深度学习

近年来，深度学习已趋于流行，并开始推动人工智能（AI）普及的革命。尽管某些技术并非全新，但海量数据和廉价计算能力使得深度学习被广泛采用。在本章中，你将学习本书剩余部分所需的深度学习基本概念和术语。

相较于硬编码规则的传统方法或算法，深度学习使机器或计算机能够从原始数据中学习并做出预测。深度学习是机器学习的分支，而机器学习本身则是 AI 的分支。深度学习技术在一定程度上受到了神经科学的启发。

### 3.1.1 感知器

我们首先介绍感知器模型，这是最简单的神经网络模型。当在标记好的训练数据集上训练时，它可以基于输入和输出来学习线性映射。线性映射是一组输入变量（即特征）权重乘积的总和。在处理分类问题时，最终的总和通过一个阶跃函数来选择 0 或者 1。感知器如图 3-1 所示。

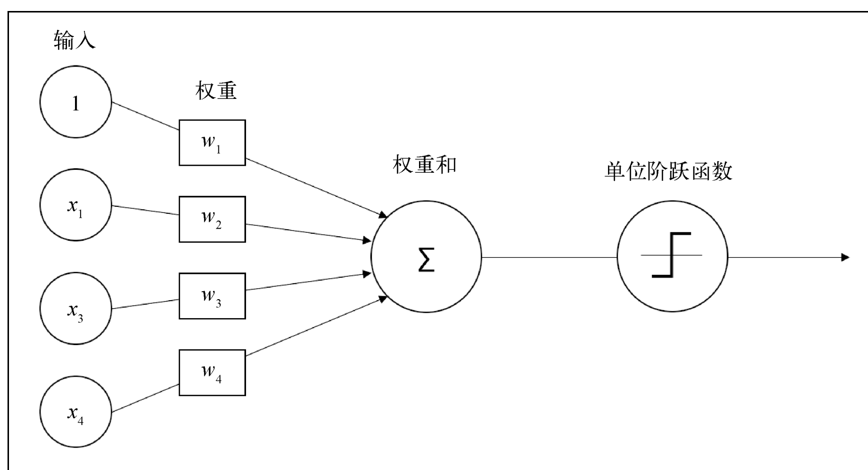



图　3-1

权重是通过学习过程从训练数据中得出的，在本章稍后将对该过程做出说明。感知器使用单位阶跃函数进行输出预测，最终激活的输出可以为 0 或 1，对应于训练数据中的二进制类别。单位阶跃函数是最简单的激活函数，下面将介绍其他几种在现代深度学习架构中广泛使用的激活函数。

## 3.1.2　激活函数

激活函数是神经网络的重要组成部分，可以将神经网络节点的输入转换为非线性输出，这使得神经网络能够从数据中学习任意非线性映射或模式。激活本身可以被视作触发系统的事件。对于之前的单位阶跃函数，感知器触发与否分别对应值 1 或 0。除此之外，还有其他的激活函数，例如 sigmoid 函数、双曲正切函数（hyperbolic tangent）和线性整流函数（rectified linear unit，ReLU）等，下面将展开讨论。

### 1. sigmoid

sigmoid 激活函数可以将任何输入转换为概率分布输出。总的来说，sigmoid 函数将任意值压缩或映射到 0 和 1 之间的值，因此被广泛用于二进制分类任务，其输出可被视为属于该类的概率。sigmoid 激活函数如图 3-2 所示。

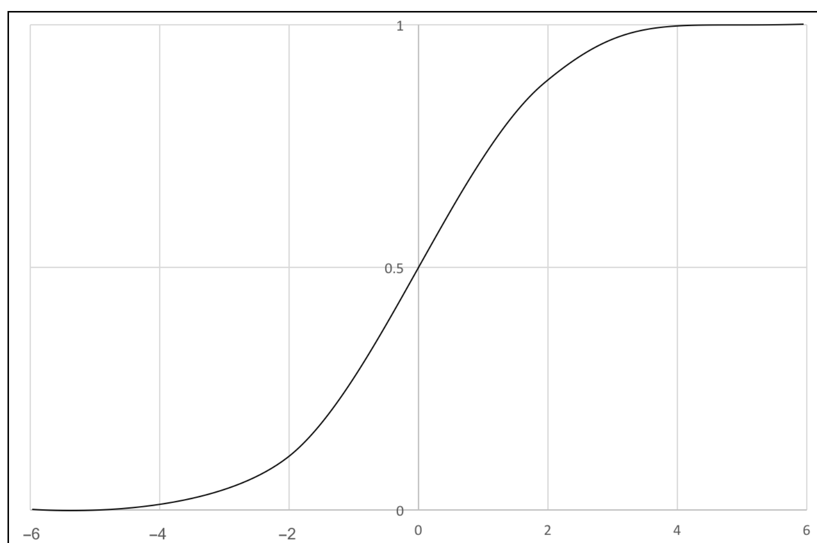



图 3-2

如图 3-2 所示，sigmoid 函数类似于单位阶跃函数，但其曲线更为平滑。这种平滑性确保了函数在整个取值范围内的差异性，而这在网络训练期间是必需的。我们将在后面对此进行讨论。

### 2. 双曲正切函数

双曲正切激活函数将输入转换为–1 和 1 之间的取值。图 3-3 是双曲正切函数的图形表示。

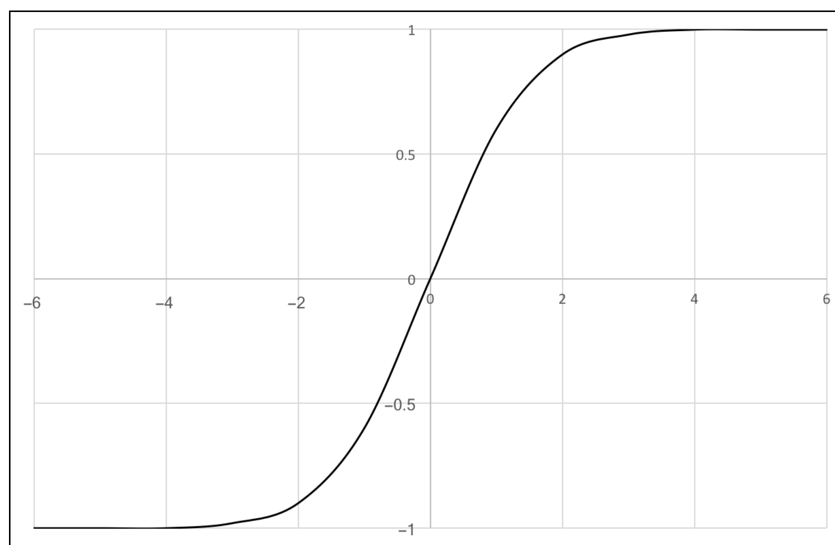



图 3-3

如图 3-3 所示，该函数类似于 sigmoid 函数，但是其 $y$ 值在−1 和 1 之间变化。双曲正切函数的主要优点是当 $x$ 取负值时，函数值不会减小。与 sigmoid 函数相比，双曲线函数在其范围内具有更高的梯度。该激活函数也称 tanh 函数。

### 3. ReLU

ReLU 限制了从负值到 0 的输入，但对于正值输入，其输出与输入相同。对于正值而言，ReLU 函数具有恒定的梯度，但对于负值而言其梯度为 0。ReLU 的图形表示如图 3-4 所示。

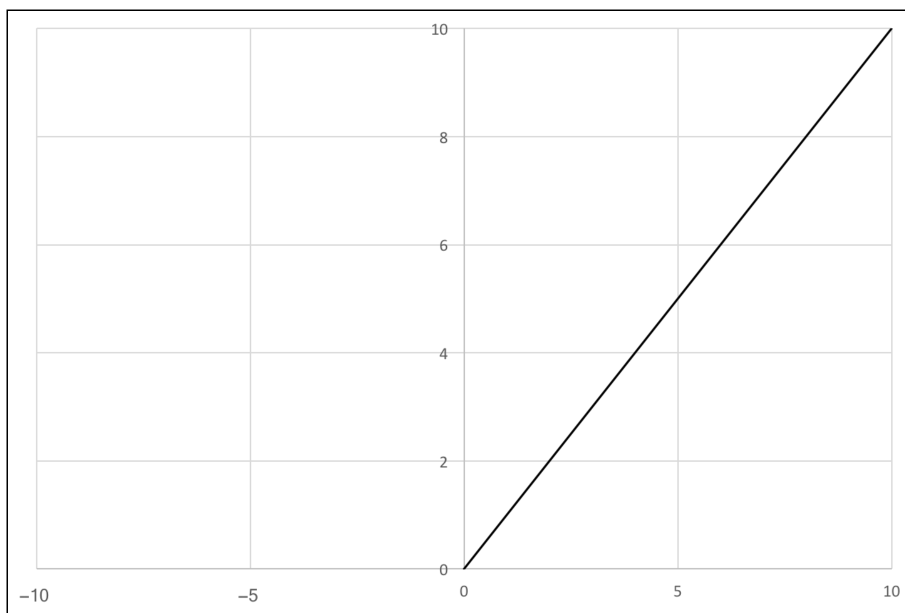



图　3-4

由此可以看出，对于负值输入，ReLU 根本不会触发。该激活函数的计算复杂度低于前述几个函数，因此其预测将更快一些。在下一节中，你可以看到如何互相连接多个感知器以形成一个深度神经网络。

## 3.1.3　神经网络

神经网络是互相连接的神经元组成的网络。一个神经网络的示例如图 3-5 所示。

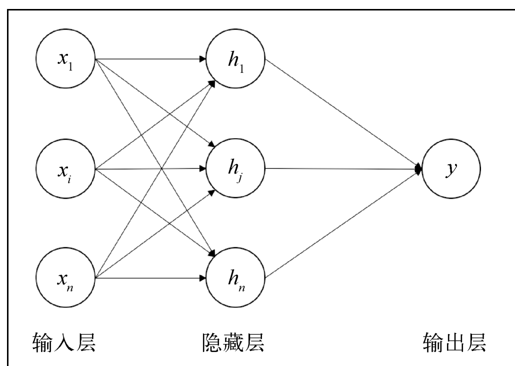



图 3-5

在图 3-5 中，输入数据被提供到输入层，然后被传递到隐藏层。隐藏层中每个单元或节点都基于输入来计算激活并将其传递到最终的输出节点。输出节点根据来自隐藏层的所有输入来计算最终输出。尽管图中仅显示了一个隐藏层，但是实际的网络可以有多个隐藏层。输出中的激活函数可用于二分类数据。如果要预测多标签类别，则还需要一些其他技术。下面我们学习独热编码（one-hot encoding）、softmax 和交叉熵（cross-entropy）。

### 1. 独热编码

独热编码是用于标记数据（尤其是分类数据）的向量化技术。对于二进制标签而言，目标变量将显示为[0, 1]和[1, 0]，对于三个类别，相同的表示形式将显示为[0, 0, 1]、[0, 1, 0]和[1, 0, 0]。独热编码支持任意数量的类别，其主要优点在于，与任意分类标注方法相比，它对所有类别数据一视同仁。例如，虽然可以使用诸如 0、1 和 2 之类的整数来表示颜色（例如红色、绿色和蓝色）的类别，但尽管颜色本身没有固有顺序，某些 ML 模型却可能会将此类输入视为按序排列的。独热编码避免了这种情况：因为分类值是二进制编码的，所以它不会假定分类值具有任何顺序。

### 2. softmax

softmax 将任意值的向量归一化或压缩到 0 和 1 之间的概率分布，其向量输出的总和等同于 1。因此，它通常用于神经网络的最后一层来预测输出类别可能的概率。以下是下标为 $j$ 向量的 softmax 函数数学表达式：

$$\text{Softmax}(z_j) = \frac{\sum e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

其中 $z_j$ 表示第 $j$ 个向量的值，$K$ 表示类别的数量。从公式可以看出，指数函数使输出值更加平滑，而分母将 0 和 1 之间的最终取值进行归一化。

### 3. 交叉熵

交叉熵是分类任务在训练期间的损失。它实际上计算了 softmax 概率或预测与真实分类之间的差异。以下是二分类问题中交叉熵的表达式，其输出用概率 $\hat{y}$ 表示，真实值用 $y$ 表示：

$$\text{CrossEntropy} = -y\log(\hat{y}) - (1-y)\log(1-y)$$

可以看出，当预测的概率接近 1 而真实输出为 0 时，交叉熵将增加或进行惩罚。同样，表达式可以扩展到 $K$ 个类别时的情况。

## 3.1.4　训练神经网络

定义网络结构及其训练方式的变量称为超参数。因为需要优化若干超参数，所以对深度神经网络的训练显得较为困难。隐藏层的数量和需要使用的激活函数都是定义架构的超参数示例。类似地，训练数据的学习率和批大小都是与训练相关的超参数示例。其他主要参数则是需要通过训练输入数据获得的网络权重和偏差。获得网络各种参数的机制或方法称为训练。

### 1. 反向传播

训练算法的目标是找到最小化某个损失函数的网络权重及偏差，而这取决于预测输出和真实标签或真实值。为此，我们在输出端计算相对于权重和偏差的损失函数的梯度，并将误差向后传播至输入层，而这些误差又被用来计算所有中间层直到输入层的梯度。这种计算梯度的技术称为反向传播。在反向传播的每个迭代过程中，当前预测结果中的误差都将通过网络进行反向传播，以针对各层权重和偏差计算梯度。

图 3-6 展示了反向传播方法。

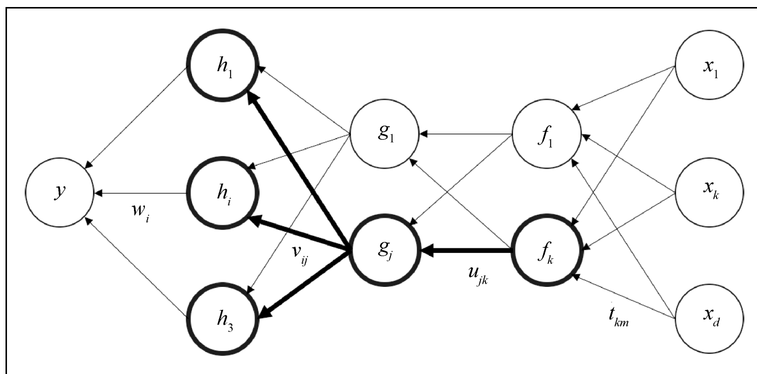



图　3-6

这种使用反向传播来更新权重及偏差的训练算法称作梯度下降，之后将进行详细说明。

### 2. 梯度下降

梯度下降是一种优化方法，它利用反向传播计算出的梯度来更新权重和偏差，不断逼近最小化损失。如图 3-7 所示，通过沿着函数的斜率或梯度调整权重，代价（损失）能达到最小值。

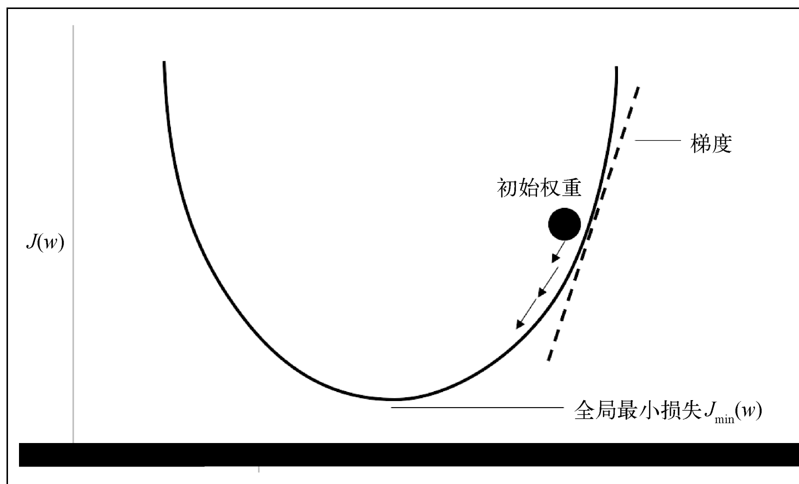



图　3-7

对于简单的感知器，损失函数与权重的关系是线性的，但对于深度神经网络，损失函数通常是多维、非线性的。由于梯度下降必须沿所有维度遍历路径，在可接受时间内达到全局最小值往往十分困难。为了避免出现此问题并加快训练速度，神经网络通常采用随机梯度下降，下面将对此进行说明。

### 3. 随机梯度下降

随机梯度下降是梯度下降算法的变体，往往用于训练深度学习模型。它的基本思想是不训练整个数据集，而是使用训练集的子集。从理论上来说，一个样本已足以训练网络。但是实际上，随机梯度下降通常使用固定数量的输入或批数据（batch）。与批量梯度下降（vanilla gradient descent）[①]相比，这种方法可加快训练速度。

### 4. 正则化技术

过拟合是机器学习中的常见问题：模型盲目地学习了数据中包括噪声在内的所有模式。由于有大量参数可用，神经网络在训练过程中很容易产生过拟合。从理论上讲，给定任意大小的输入数据，足够大的人工神经网络（artificial neural network，ANN）都可以记住其中的所有模式及噪

① 也叫 batch gradient descent。vanilla 是指标准、常规或未修改的版本。vanilla 梯度下降法是指基本的批量梯度下降算法，未经过优化和修改。——译者注

声。因此，我们必须对模型的权重进行正则化处理以避免数据的过拟合。

我们将学习三种正则化技术：

- 随机失活（dropout）
- 批标准化
- L1 和 L2 正则化

- **dropout**

dropout 机制是在训练过程中暂时丢弃某些神经元来达到正则化的技术。因此，权重是标准化的。图 3-8 显示了一个神经网络，左侧为标准网络，右侧为 dropout 后的网络。

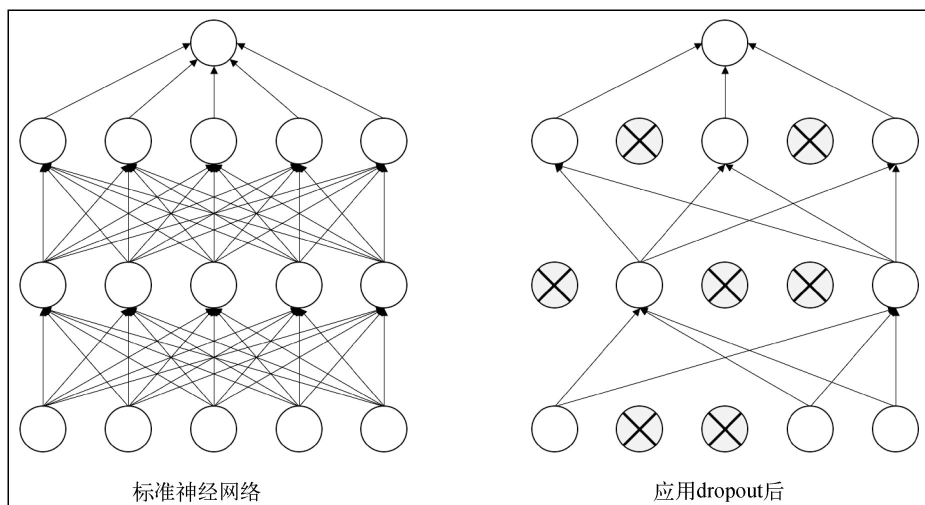



图　3-8

实际上，dropout 可防止网络过分重视可能会导致过拟合的任意单个节点或特征。因此，使用 dropout 将权重值分布在不同的节点上可以对输出实现正则化。另一种对数据本身起作用的正则化技术是批标准化，下面将进行说明。

- **批标准化**

批标准化可重新调整网络的输入和所有中间层输出的大小，使训练过程更流畅、迅速。在进行缩放后，所有输入和输出的平均值均为 0、标准差为 1。这有助于神经网络提升训练速度，并带来正则化的效果。

- **L1 和 L2 正则化**

L1 和 L2 正则化是常见的正则化技术，可控制训练过程中网络权重增长或收缩的幅度。与

dropout 类似，它使网络不会对部分特征的过分重视。在 L1 正则化中损失函数与权重的大小成正比，而在 L2 正则化中损失函数与权重的平方成正比。

### 3.1.5　卷积神经网络

卷积神经网络（CNN）通过学习卷积核来对数据进行变换。卷积使得变换对于平移保持不变。随着层层加深，特征的深度会根据分配的过滤器数量而变化。图 3-9 对此做出了说明。



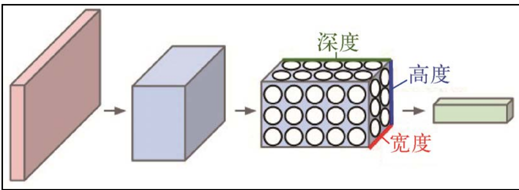



图　3-9

#### 1. 核

核是 CNN 用来在特征图（feature map）上滑动的小窗口。如图 3-10 所示，核进行滑动窗口运动并生成输出的特征图。

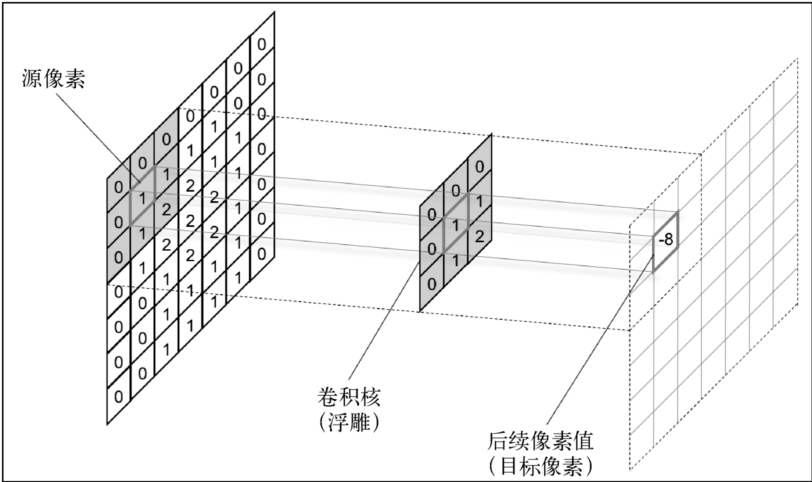



图　3-10

核可以选用不同大小的矩形并以较长或较短的步长进行移动。

#### 2. 最大池化

最大池化（max pooling）是一种从小窗口中选取最大值的二次抽样形式，如图 3-11 所示。

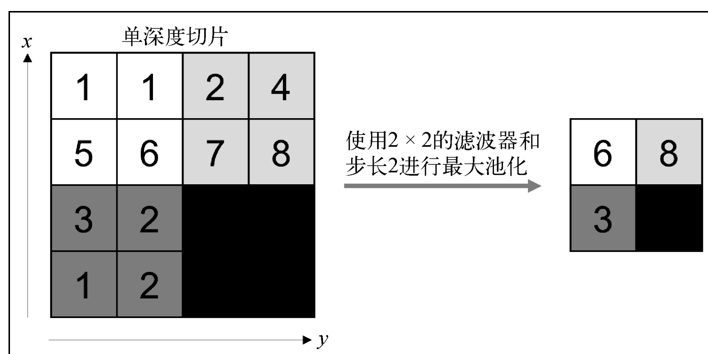



图　3-11

在图 3-11 中，最大池化从窗口中分别选取了各个小窗口的最大值。

### 3.1.6　递归神经网络

递归神经网络（RNN）可以训练语言等存在时序依赖的模型。实际上，它可以用于训练任何种类的序列数据。在 RNN 中，神经元的输出会被作为下一时刻的输入反馈到其自身。展开后的 RNN 如图 3-12 所示。

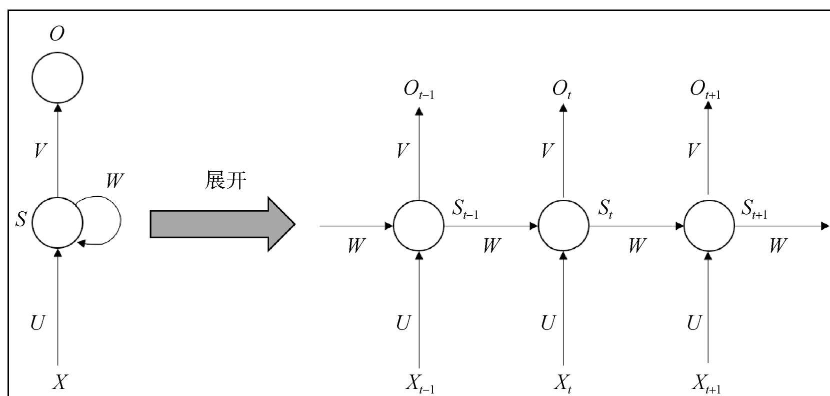



图　3-12

由于 RNN 从先前时间步中获取输入的性质，其中很久以前序列中的数据将会丢失。

#### 长短期记忆网络

长短期记忆网络（LSTM）可以通过使用“遗忘门”来记住很久以前的事。相对于 RNN 而言，LSTM 优点在于可以长时间保持记忆。LSTM 中有好几个门，诸如遗忘门、输出门和输入门，每个都有自己的功能，如图 3-13 所示。

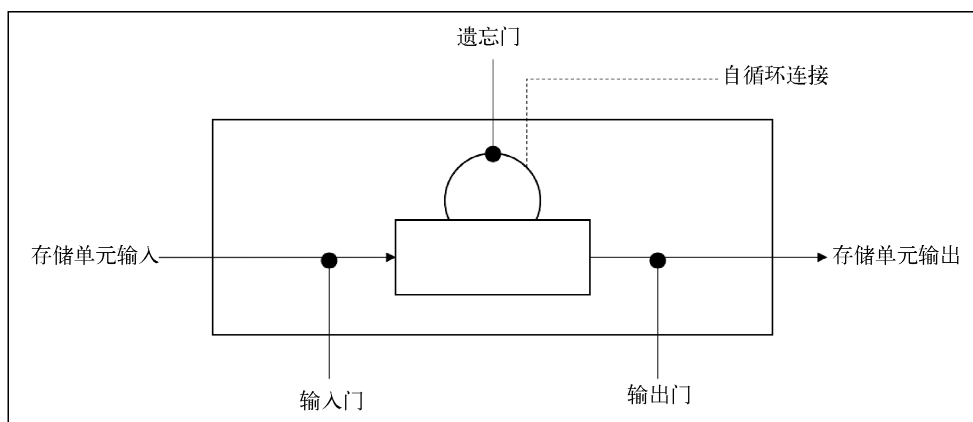



图　3-13

我们在本节学习了一些深度学习的技术，下面将了解一下 TensorFlow。

## 3.2　TensorFlow

TensorFlow 是 Google 为机器学习提供的库。它运行快、可扩展，并且具有用于可视化和部署的多种工具，已在开发者社区中普及开来并被多个组织所使用。在本节中，你将看到它的硬件要求、安装过程和几个简单的使用示例。

### 3.2.1　通用图形处理单元

通用图形处理单元（general purpose – graphics processing unit，GPGPU）可以在很大程度上加快深度学习模型训练和推理的速度。大数据和廉价计算力不断推进深度学习取得新进展。NVIDIA 公司提供了 GPU 和一些库来加速深度学习。使用 GPU 硬件进行训练很有帮助，但并不是必需的。

#### 1. 统一计算设备架构

NVIDIA 提供的统一计算设备架构（Compute Unified Device Architecture，CUDA）库将为 GPU 安装所需要的驱动程序，其下载页面如图 3-14 所示。

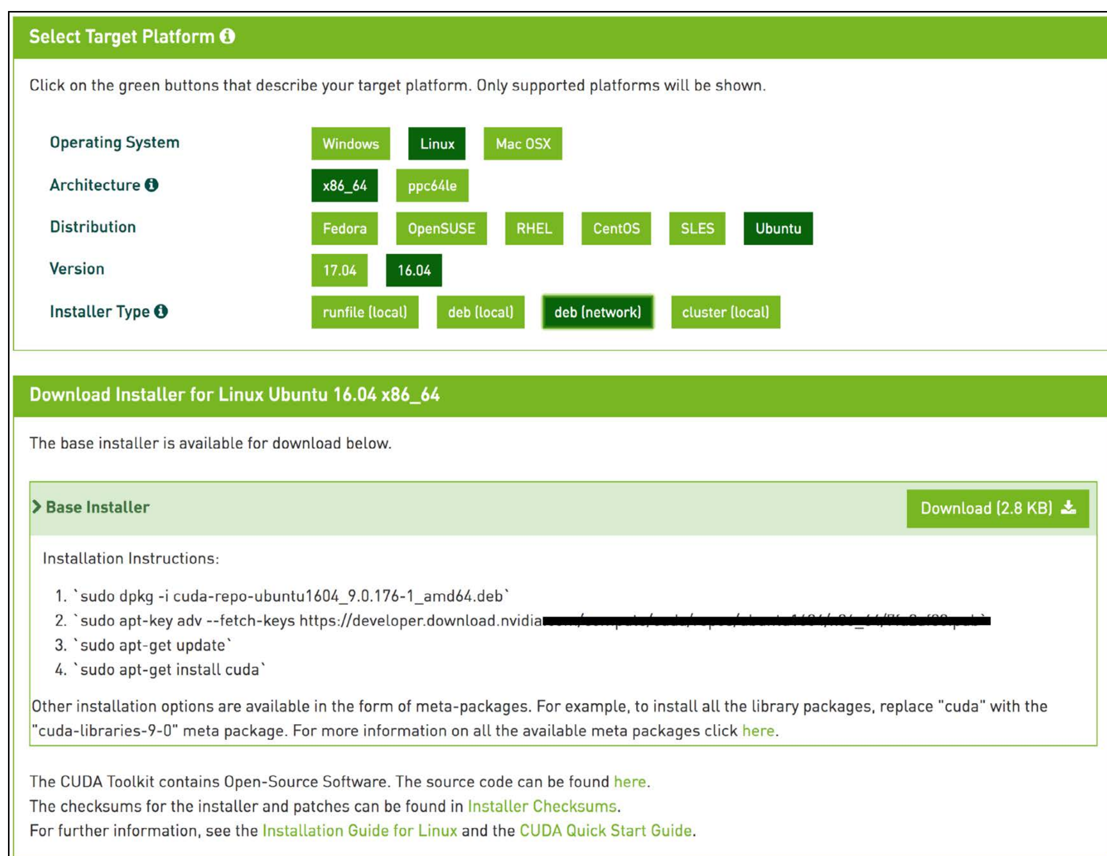



图　3-14

根据你所使用的计算机环境，根据指示选择并安装相应的版本。

### 2. CUDA 深度神经网络

当 NVIDIA 硬件投入使用时，CUDA 深度神经网络（CUDA Deep Neural Network，cuDNN）可被用来加速深度学习模型的训练和推理过程。

## 3.2.2　安装

在安装有 Python 的环境中，TensorFlow 的安装非常简单。根据是否具有 GPU，可以使用以下代码进行安装：

```
sudo pip install tensorflow
```

也可以用以下代码：

```
sudo pip install tensorflow-gpu
```

接下来，我们将会看到一系列示例。

### 3.2.3　Hello world！

下面来看看如何用 TensorFlow 完成 `Hello, world!`的输出。我们将直接在 Python shell 中完成这一示例：

(1) 进入 Python shell。

(2) 使用以下代码导入 TensorFlow：

```
>>>import tensorflow as tf
```

(3) 通过基于 `tf` 的定义，TensorFlow 可以理解并使用常量、变量和操作等。使用以下代码定义一个常量字符串：

```
>>>hello_world = tf.constant("Hello, world!")
```

(4) 创建一个会话：

```
>>>sess = tf.Session()
```

(5) 运行会话：

```
>>>print(sess.run(hello_world))
```

(6) 输出应如下所示：

```
Hello, world!
```

恭喜！你已经写出了首个使用 TensorFlow 的程序。下面我们将介绍使用 TensorFlow 的一些概念。

### 3.2.4　两数相加

现在来看看使用 TensorFlow 完成两数相加的示例。占位符是 TensorFlow 图中的节点，而会话指的是图初始化并准备处理值的时刻。在会话期间，我们可以将值传递到占位符中。

(1) 我们将定义两个用于存储整数的占位符：

```
a = tf.Placeholder(tf.int32)
b = tf.Placeholder(tf.int32)
```

(2) 将两个变量相加并存储在新变量中：

```
c = a + b
```



注意，这个操作仅仅被定义好但尚未执行。

(3) 创建值字典以加载到占位符。字典的键本身就是带有占位符的变量，如下所示：

```
values = {a: 5, b: 3}
```

(4) 创建一个会话。当会话启动时，TensorFlow 图会被载入到内存，准备好将值传入占位符中并进行处理：

```
sess = tf.Session()
```

(5) 将值传入图中并运行会话。我们需要返回节点 c 的输出，所以使用以下代码：

```
print(sess.run([c], values))
```

输出结果应为[8.0]。这个例子展示了数值被馈送到图中并处理的过程。

### 3.2.5 TensorBoard

接下来使用 TensorBoard 进行图的可视化工作。我们要更新加法程序，使其包含 TensorBoard 的说明。可以为任何节点分配名称，以便在 TensorBoard 中使用相应的名称对其进行渲染：

(1) 在以下代码片段中，给了占位符分配了名称'a'、'b'和 'c'：

```
a = tf.Placeholder(tf.int32, name='a')
b = tf.Placeholder(tf.int32, name='b')
c = tf.add(a, b, name='add')
values = {a: 5, b: 3}
sess = tf.Session()
```

(2) 创建值并开始会话，在此之后将文件路径作为参数创建摘要编辑。摘要所需的详细信息将被存储在该文件中，并可用于显示 TensorBoard：

```
summary_writer = tf.summary.FileWriter('/tmp/1', sess.graph)
```

(3) 如上节中一样运行会话：

```
sess.run([c], values)
```

(4) 当程序开始运行时，进入命令提示符，使用摘要文件的路径作为参数并键入以下命令：

```
tensorboard --logdir=/tmp/1
```

(5) 进入以下链接：

```
http://localhost:6006/
```

你将看到如图 3-15 所示的界面。

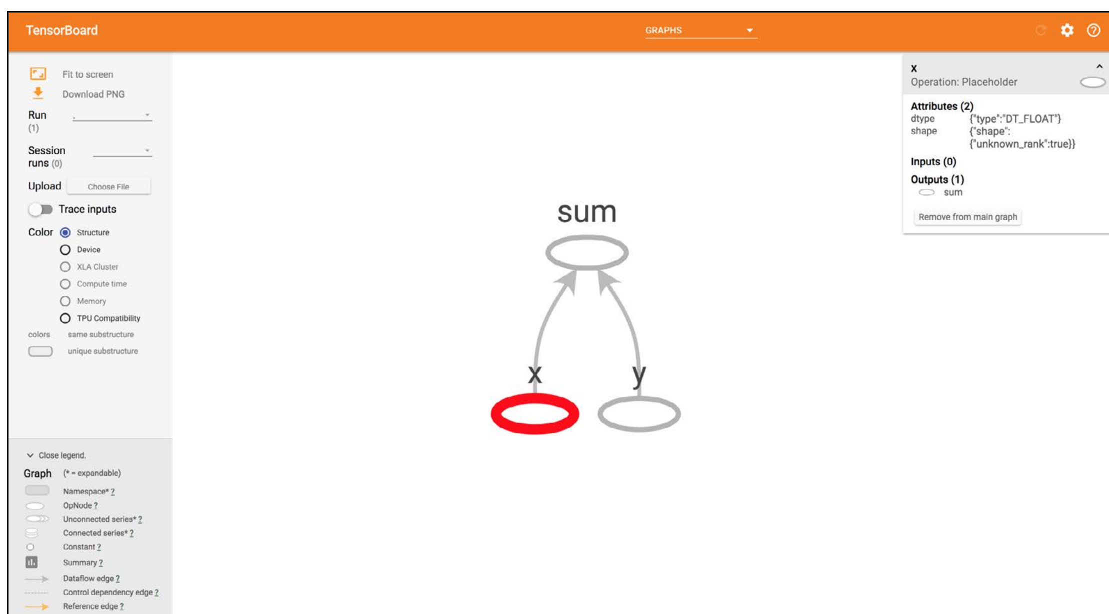



图　3-15

在随后的章节中，你将学到 TensorBoard 的其他用途。

### 3.2.6　Keras库

Keras 库是一个可以在后端使用多个深度学习库的简化 API。由于用法简单，Keras 在深度学习社区中很受欢迎。Keras 可以在后端使用 TensorFlow 或 Torch 框架。本书将以 TensorFlow 和 Keras 为例。此外，TensorFlow 在 `tf.keras` 模块下还具有 Keras 版本。

## 3.3　小结

在本章中，你学习了深度学习和 TensorFlow 的基础知识。本章涉及的词汇将在全书范围内使用。诸如 CNN 和 LSTM 之类的概念是许多应用的基本构建模块。

在下一章，我们将使用 LSTM 为不同的应用训练一个深度学习模型以进行文本分类。

第 4 章



# 使用浅层模型进行语义嵌入

本章将探讨理解单词间语义关系的动机，并讨论识别语义关系的方法。在此过程中，我们将获得单词的向量表示，这有助于构建文档级别的向量表示形式。

本章涵盖以下主题：

- 由简单浅层神经网络模型训练的词嵌入（word embedding），它可将词表示为向量
- 连续词袋（CBOW）嵌入，它使用类似的神经网络训练，可以从给定词预测目标
- 通过平均 Word2vec 得到的句嵌入
- 通过文档平均获得的文档嵌入

## 4.1 词向量

词向量（word vector）是许多应用中非常有用的构建模块。它可以对词间的语义关系进行捕获和编码，并最终将单词转换为数字序列，从而形成非常适合训练深度学习模型的密集向量。本章将详细介绍各种方法，用于构建这种便于分析语义的词嵌入。

### 4.1.1 经典方法

构建单词表示的传统方法一般使用词袋模型。在该模型中，词表示将各个单词视为彼此独立的。因此此类表示通常使用独热编码生成句子或文档的向量表示，以显示句子中单词的存在与否。但这种表示在实际应用中鲜有使用，因为单词的含义会根据周围单词而变化。例如，考虑句子“The cat sat on the broken wall”以及“The dog jumped over the brick structure”。可以明显地看出，尽管这两个句子讨论的是两个独立的事件，但是其语义是相似的。例如，dog 与 cat 都属于一个称为 animal 的实体，所以是相似的；而 wall 则可以被视作类似于 brick structure。因此，尽管两个句子分别讨论了不同的事件，但它们在语义上是相互关联的。在使用词袋模型（其中，单词以

其自身维度进行编码）的经典方法中，实际上无法对这种语义上的相似性进行编码。

看看下面的句子：

- TensorFlow is an open source software library
- Python is an open source interpreted software programming language

如果我们认为这两句话分别属于独立的文档，则可以创建两个词列表：

- [TensorFlow, is, an, open, source, software, library]
- [Python, is, an, open, source, interpreted, software, programming, language]



此时，先前两个文档的词汇可以合并写成：[TensorFlow, is, an, open, source, software, library, Python, interpreted, programming, language]，其中包含 11 个单词。

因此，我们可以用如下形式来表示最初的文档：

- [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
- [0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1]

在上述表示中，每个数字表示词汇列表中对应位置的单词在文档中重复出现的次数。由此可以看出：当词汇量增加时，其中绝大多数单词将不会出现在每个文档中，从而使向量表示成为一个长且几乎为空（零）的形式。无论句子的长度如何，其大小将固定是词汇表的大小。

传统方法的另一个不足是无法体现单词在句子中出现的顺序。传统的词袋方法统计文档中文本的词汇量，以获得存在单词的表示形式，但这丢了失上下文。与前面讨论的编码类似，它假定文档中的单词彼此独立。这种方法还有一个局限：会导致数据稀疏，使得统计模型的训练变得更加困难。当然，这也形成了用向量表示单词的基本动机：将单词的语义编码在其表示中。

### 4.1.2 Word2vec

单词的向量表示可以实现语义相似单词的连续表示，其中相关的单词会被映射到高维空间内彼此靠近的点上。这种单词表示方法基于以下事实：有相似上下文的单词也有相似的语义。Word2vec 就是这样的一种模型，它试图通过使用相邻的单词来直接预测单词并学习小且密集的向量（也称为**嵌入**）。Word2vec 可从原始文本中学习词嵌入，是一种在计算上很有效率的无监督模型。为了学习这些密集向量，Word2vec 有两种形式：连续词袋（CBOW）模型和跳字（skip-gram）模型（由 Mikilov 等提出）。

Word2vec 是一个浅层的三层神经网络，其中第一层和最后一层构成输入和输出，中间层构建潜在表示以便将输入单词转换为输出向量表示形式。

Word2vec 单词表示法可以探索词向量之间有趣的数学关系，这也是单词的一种直观表达。例如，我们能够通过使用单词表示来找出该表达式的值：

$$\text{king} - \text{man} = \text{queen} - \text{woman}$$

在数学上，此表达式得到的是所求词向量值在潜在空间的等价性。另一方面，从直觉上来说，我们可以理解为：从 king 中删除 man 并增加 woman 会得到 queen。仅当利用单词的位置关系从而理解上下文时才能建立这种等式关系。从语义上可以明显看出，“国王”一词与男人同处一个位置，与“女王”和“女人”一词的出现方式类似，如图 4-1 所示。

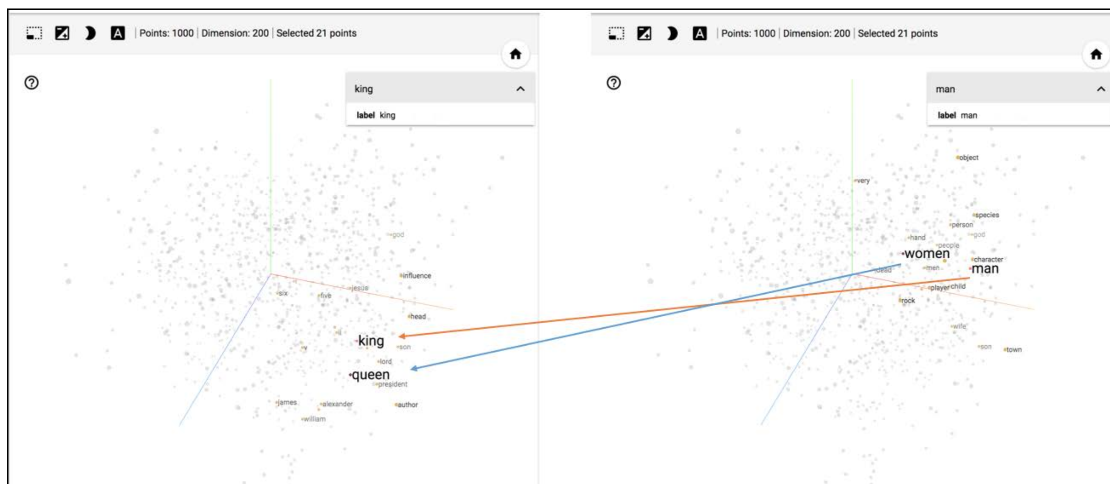



图 4-1　词向量转换

图 4-1 表明词向量是如何从 woman 转换到 queen 的，这与从 man 转换到 king 具有相似之处。使用 Word2vec 可以理解该关系，该模型使用了一个简单的三层神经网络来预测周围的单词（给定输入单词）或预测该单词（给定周围的单词）。这两种方法都是 Word2vec 的变体，其中使用输入单词来预测周围单词的方法是跳字模型，而使用周围单词来预测目标单词是连续词袋模型。

## 4.1.3　连续词袋模型

Word2vec 的连续词袋模型从一组输入源的上下文单词来预测目标单词。这意味着在句子“The cat sat on the dirty mat”里，连续词袋尝试通过使用上下文单词 the、cat、sat、on 和 dirty 来预测 mat 的目标词向量。为了实现这一点，连续词袋构建了一个上下文目标单词对的元组。因此，对于一组上下文词汇（the、cat、sat、on 和 dirty），我们要预测单词 mat。这是通过 `(the, mat)`、`(cat, mat)`、`(sat, mat)`、`(on, mat)`和`(dirty, mat)`实现的。

### 4.1.4 跳字模型

跳字模型执行连续词袋任务的逆操作，通过使用目标单词来预测上下文中的相邻单词。以先前讨论的示例“The cat sat on the dirty mat”为例，跳字尝试使用 mat 的词向量来预测 cat、sat、on 和 dirty 的目标词向量。因此，对于背景词 mat，我们预测目标词（the、cat、sat、on 和 dirty）。这是通过`(mat, the)`、`(mat, cat)`、`(mat, sat)`、`(mat, on)`和`(mat, dirty)`表示的。

#### 1. 跳字模型和连续词袋模型的架构比较

图 4-2 展示了跳字和连续词袋模型架构的比较。

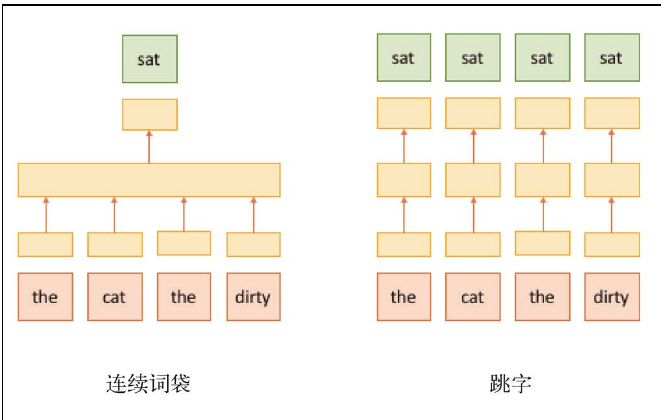



图 4-2

图 4-2 显示了跳字如何通过使用上下文中的单词来学习预测目标单词，而连续词袋会根据目标单词周围固定大小窗口的单词（以词袋表示）来学习预测目标单词。

通常，当数据集更大时，跳字方法倾向于产生更好的单词表示。因此我们在本章的剩余部分将侧重于建立一个跳字模型。同时，我们还将研究使用 TensorBoard 对训练好的词嵌入进行可视化，这将使我们能够理解词嵌入的原理。下一节，我们将遍历代码并分析结果。

#### 2. 建立一个跳字模型

首先导入示例所需的 Python 模块：

```
from tensorflow.contrib.tensorboard.plugins import projector

import os
import numpy as np
import tensorflow as tf
```

TensorFlow 的 projector 模块为我们提供了在 TensorBoard 上添加词向量以进行可视化所需的方法。随后我们将创建一个字典，其中将包含用于训练 Word2vec 模型的所有模型参数：

```
# 有关模型训练的参数

model_params = {
    "vocab_size": 50000,    # 最大单词数
    "batch_size": 64,       # 各个训练步的批大小
    "embedding_size": 200,  # 词嵌入向量维度
    "num_negatives": 64,    # 否定词采样数
    "learning_rate": 1.0,   # 训练学习率
    "num_train_steps": 500000, # 模型训练步数
}
```

我们将定义 Word2vecModel 类，用于模型的定义、训练及可视化例程。该类及其__init__方法如以下代码所示：

```
class Word2vecModel:
    """
    为 Word2Vec 模型初始化参数
    """
    def __init__(self, data_set, vocab_size,
                 embed_size, batch_size, num_sampled, learning_rate):
        self.vocab_size = vocab_size
        self.embed_size = embed_size
        self.batch_size = batch_size
        self.num_sampled = num_sampled
        self.lr = learning_rate
        self.global_step = tf.get_variable('global_step',
                                           initializer=tf.constant(0),
                                           trainable=False)
        self.skip_step = model_params["skip_step"]
        self.data_set = data_set
```

我们将使用__init__方法来初始化 Word2vec 模型参数。如先前所示，我们将使用该函数初始化模型的学习率、批大小、词汇大小及嵌入向量大小。之后利用 TensorFlow 中的 Dataset API 生成器导入数据，如下所示：

```
data_set = tf.data.Dataset.from_generator(generator,
                                          (tf.int32, tf.int32),
                            (tf.TensorShape([model_params["batch_size"]]),
                             tf.TensorShape([model_params["batch_size"],
1])))
```

我们使用 Dataset API 从生成器生成样本，并使用数据集的 from_generator 方法来生成数据（其元素由生成器生成）。generator 的参数应为可调用对象并返回支持 iter() 协议的对象。这可以是一个 generator 函数。生成器生成的元素必须与给定的 output_types 参数以及可选的 output_shapes 参数相兼容。我们可以编写一个生成器方法，如以下代码所示：

```
def generator():
    yield from batch_generator(model_params["vocab_size"],
                               model_params["batch_size"],
                               model_params["skip_window"],
                               file_params["visualization_folder"])
```

我们将定义一个方法用于导入专门为其创建 generator 的数据。为保证 TensorFlow 图对于 Python 操作定义完好，我们将使用 TensorFlow 的 name_scope：

```
with tf.name_scope('nce_loss'):
    # 为 NCE 损失构建变量
    nce_weight = tf.get_variable('nce_weight',
                        shape=[self.vocab_size, self.embed_size],
                        initializer=tf.truncated_normal_initializer(
                        stddev=1.0 / (self.embed_size ** 0.5)))
    nce_bias = tf.get_variable('nce_bias',
                    initializer=tf.zeros([model_params["vocab_size"]]))

    # 将损失函数定义为 NCE 损失函数
    self.loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weight,
                                    biases=nce_bias,
                                    labels=self.target_words,
                                    inputs=self.embedding,
                                    num_sampled=self.num_sampled,
                                    num_classes=self.vocab_size),
                               name='loss')
```

之后创建另一个 name_scope，以初始化嵌入矩阵和嵌入查找（embedding lookup），后者可以检索数据集中任意给定单词的嵌入。然后再创建一个 name_scope 来定义损失函数：使用**噪声对比估计**（noise contrastive estimation，NCE）损失将多项式分类问题（例如预测下一个单词的问题）转换为二进制逻辑回归问题。

对于数据集中的每个训练样本，增强型分类器都将获得一个真值对（一个值出现在中心词上，另一个值出现在中心词的上下文中）和 $k$ 个随机选择的否定对（由中心词和不在所选单词上下文中的随机单词组成）。通过学习区分真值对和否定对，分类器就学习到了词向量。实际上，这种损失确保了优化的分类器可以预测一对单词的好坏，而非预测下一个出现的单词。下面，我们将定义用于训练的优化器：

```
self.optimizer =
  tf.train.GradientDescentOptimizer(self.lr).minimize(self.loss,
                                        global_step=self.global_step)
```

GradientDescentOptimizer 是实现了梯度优化算法的 optimizer 方法，它允许设定学习率参数及优化执行的步长。

最终，我们将使用损失值建立直方图及标量摘要以监测训练过程中的损失变化，同时合并所有摘要以在 TensorBoard 上展示：

```
with tf.name_scope('summaries'):
    tf.summary.scalar('loss', self.loss)
    tf.summary.histogram('histogram loss', self.loss)
    self.summary_op = tf.summary.merge_all()
```

在这些步骤中，我们训练神经网络进行 `train_steps` 并监测损失。训练的目的通常是降低损失，但如果训练数据较少、训练时间较长并使用更高维度的单词表示形式，通常会发生过拟合。因此需要确保模型不会对训练数据产生过拟合，还需确保模型具有很好的泛化能力。

在下一节，我们将在 TensorBoard 中将嵌入映射到低维中以完成可视化。

### 3. 词嵌入的可视化

我们将训练后的词向量嵌入 TensorBoard 并通过将训练好的向量投影到二维空间来实现可视化。要生成此类投影，可以使用 TensorBoard 中的 t-SNE 或 PCA 方法。图 4-3 显示了使用 PCA 生成的投影在 TensorBoard 上的结果。

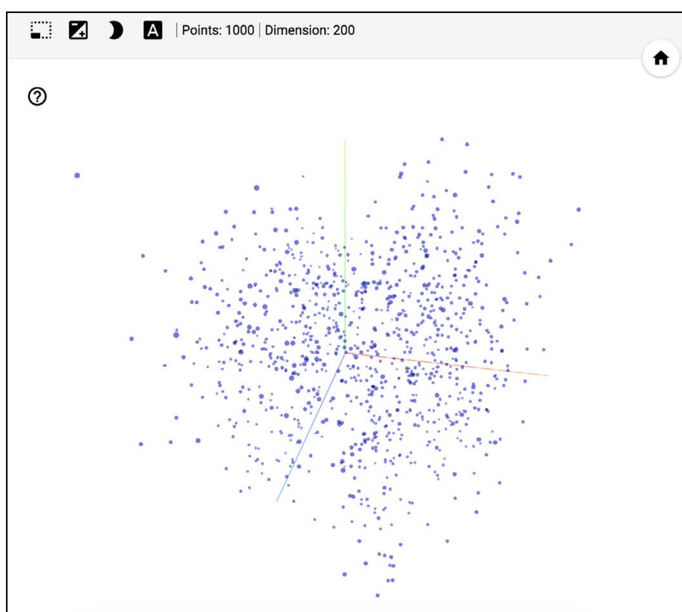



图　4-3

该可视化图说明了 TensorBoard 如何通过投影显示嵌入，但这种可视化效果看起来用处不大，而且使用 PCA 降维使其结果在查看时毫无意义。因此我们将可视化模式切换为使用 t-SNE 进行投影，这是另一种非常适合可视化高维数据的降维技术，结果如图 4-4 所示。

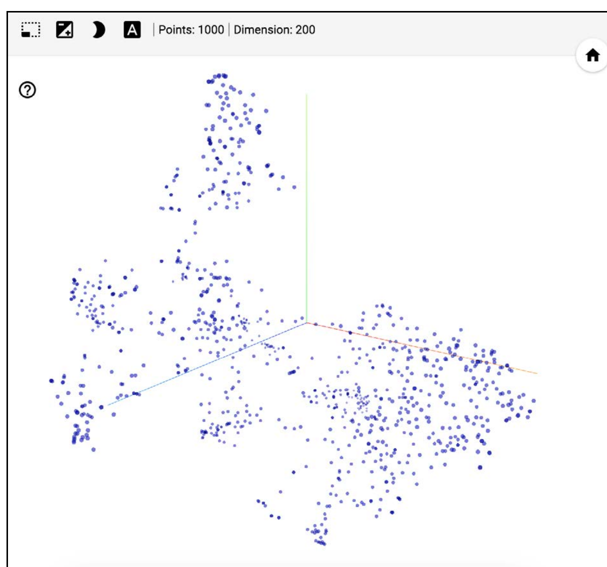



图 4-4

从 t-SNE 投影图像不同区域中出现的簇可以看出，词嵌入似乎已具有某些模式。为了了解这些簇所发现的主题，TensorBoard 允许有选择地放大部分区域并查看其中的基础数据，操作如图 4-5 所示。

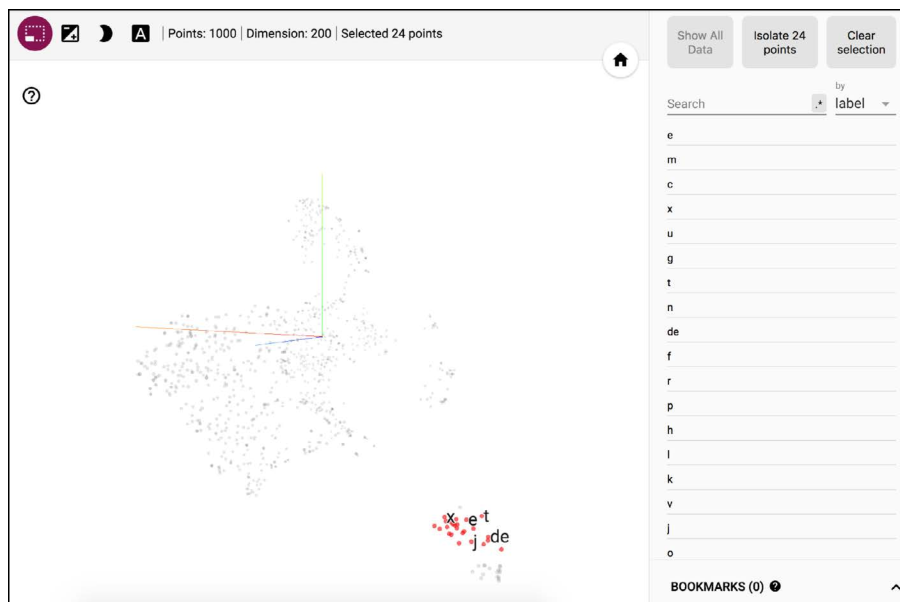



图 4-5

当我们在 TensorBoard 上检查一个孤立的簇时，可以明显看到这些向量已捕获了一些有关单词及其相互关系的常规语义信息。有趣的是，所选特定类别中的单词只有一个英文字符的长度。

现在我们将搜寻一个特定的单词并查看它最近的邻居，如图 4-6 所示。

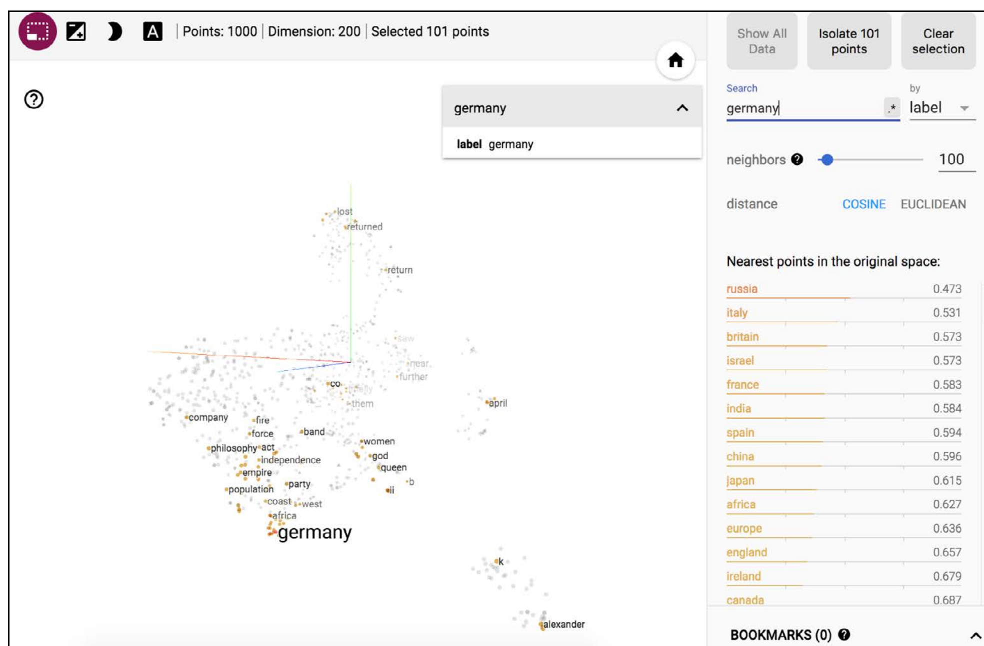



图 4-6　germany 一词的簇板

在本例中，我们搜寻单词 germany 并发现最靠近给定单词的是 russia、italy 和 britain 等。这些都是国家的名字。有趣的是，我们从未以标注或任何其他形式向模型提供这些信息。

关于模型发现词间语义关联的另一个示例如图 4-7 所示。

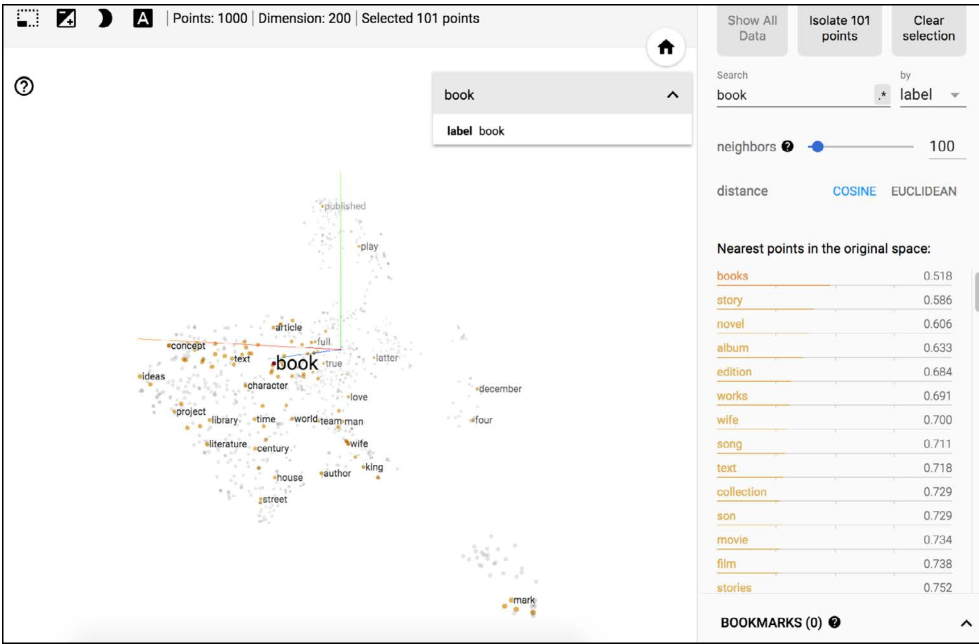



图 4-7 book 一词的簇板

这个例子表明最靠近单词 book 的是 story、novel 和 album。这种语义关联被模型发现佐证了即使没有提供先决条件，Word2vec 也大有用处。

## 4.2 从单词到文档嵌入

Word2vec 提供了一种生成恰当词向量的优雅方法。然而因为每个文档中单词的数量是可变的，句子或文档级别的向量表示对于词向量来说并不总是存在的。因此将词嵌入扩展到文档嵌入的最简单方法之一是对文档中可用的各个词嵌入进行平均。

因此，文档嵌入可用以下公式表示：

$$\mathrm{Emb}(d)=\sum_{i=1}^{m} w_i \times \mathrm{Emb}(w_i)$$

在该公式中，为了获得最终的文档嵌入，我们给予句子中所有单词以相同的权重，因此有 $w_i = 1/m$。然而这种方法假定文档中的所有单词对于表达文档的含义都具有相同的贡献。

## 4.3 Sentence2vec

前面所讨论方法（用于获取文档级向量表示）的主要缺点之一是文档中出现的单词权重相等，

而这抑制了句子中特定单词的重要性。实际上，这些单词为文档增加了含义和价值。我们用该方法构建文档向量表示的具体示例来对这一点进行讨论，看看“Jack and Jill went up the hill to fetch a pail of water”。在该句子中，提供与所发生动作有关的上下文信息最多的单词是 Jack、Jill、hill、fetch、pail 和 water。但是先前的方法将对句子中的每个单词赋予相等的权重。

之前讨论过的一种有趣方法是使用词向量的加权平均值，而非对单词进行平均加权。计算单词权重的一种方法是利用文档中存在的标记使用每个单词的 TF-IDF 权重。这种方法背后逻辑在于：文档中频繁出现的单词并不能传达太多信息，而不同文档中频繁出现的单词传达了有关该单词重要性的更多信息。

TF-IDF 权重可用以下公式表示：

$$tf-idf_{t,d}=tf_{t,d}\times idf_t$$

在该公式中，$tf-idf_{t,d}$基于项 $t$ 及文档 $d$ 表示了权重，如下所示：

- 当项 $t$ 仅在语料库的少许文档中出现时，其权重值最高；
- 当该项在同一文档中出现多次或者在多个文档中出现（导致单词的弱相关）时，其权重值会降低；
- 当该项在所有文档中均出现并在每个文档中出现多次时，其权重值将达到最低。

因此，得到良好文档级别表示的方法在于对文档中的各个单词进行权重归一化，并用归一化后的值衡量单词的权重。这种方法不仅对于文档表示十分有效，而且因为忽略了低权重单词，所以计算成本也很低。与先前方法类似，我们可以用下式表达 Sentence2vec 嵌入：

$$\mathrm{Emb}(d)=\sum_{i-1}^{m}w_i\times\mathrm{Emb}(w_i)$$

两者的主要区别在于，现在权重 $w_i$通过 TF-IDF 进行了计算，并发生了改变。最终的嵌入将是词嵌入向量的加权平均。

使用原始权重还有一些替代方法，包括对权重应用阈值，从而将权重转换为二进制格式。因此，阈值越低，用于产生文档表示的单词数量就越多，也就需要更多计算。与此相反，高阈值权重将显著减少用于表示文档的单词数量，从而降低计算成本，但这可能会忽略有助于文档表达的单词。

## 4.4　Doc2vec

Mikilov 等人提出了 Word2vec 模型的一个简单扩展，并将其应用于文档级别。在这种方法中，唯一的文档 ID 被附加到文档中以获得文档向量。文档中的单词参与训练，以生成词嵌入的平均（或串联），并最终生成文档嵌入。因此，在之前讨论的示例中，Doc2vec 模型数据如下所示：

- TensorFlow is an open source software library
- Python is an open source interpreted software programming language

与先前的方法相反，文档列表现在如下所示：

- [DOC_01, TensorFlow, is, an, open, source, software, library]
- [DOC_02, Python, is, an, open, source, interpreted, software, programming, language]

此 Doc2vec 模型看起来与我们在连续词袋中讨论的方法非常相似。因此当我们训练词向量 `W` 时，将同时训练每个文档的文档向量 `D`。当训练完成后，就可以同时得到词向量与文档向量。该方法与连续词袋的相似之处在于当给定上下文单词时模型会尝试预测目标单词。因此，在本例中，模型会使用 DOC_01、TensorFlow、is 和 an 来预测 open。该模型也称为**段落向量分布式内存**（paragraph vector-distributed memory，PV-DM）。文档向量背后的思想是代表文档中所讨论主题的语义上下文。

与 Word2vec 相似，PV-DM 模型的一种变体是**段落向量–分布式词袋**（paragraph vector-distributed bag of words，PV-DBOW），它类似于 Word2vec 的跳字模型。在该版本中，模型在给定上下文单词的情况下尝试预测目标词向量。例如，模型使用 DOC_01 来预测单词 TensorFlow、is、an 和 open。Doc2vec 相对于 Word2vec 的一项潜在优势在于无须存储词向量。

## 文档嵌入可视化

让我们像之前一样进行可视化，看看 Doc2vec 如何训练文档且最终的文档嵌入如何使理解文档潜在主题成为可能，如图 4-8 所示。

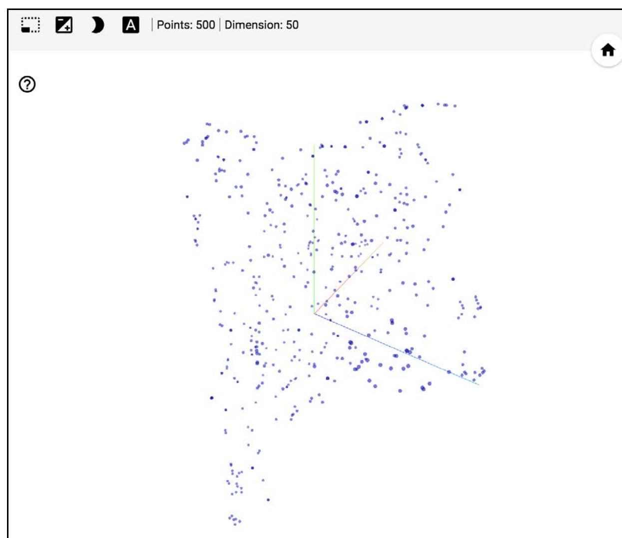



图 4-8

在该可视化图中可以看到，左上角和底部有一些可见簇。这表明我们无须这些主题的详细标注就可以就指出文档的主题。下面看看这些簇包含的内容，如图 4-9 所示。




图　4-9

在上面的簇中可以看到 Doc2vec 发现了正在讨论 malware attacks 的簇。中间的可视化部分指出了该簇，而右边的列表则显示了文档及其与 “is this government attempt to destroy cryptocurrency” 的相似性得分。尽管文档本身并不包含 malware 或 attack 等任何单词，但 Doc2vec 通过使用上下文（ransomware 在文档中的出现）依然发现该文档在讨论 malware attacks。下面来探索嵌入投影器展示的另一个簇，如图 4-10 所示。




图　4-10

该簇展示了语料库中与其他文档完全隔离开来的部分文档。这些文档探讨了保存它们的URL 并且包含 URL 和文档保存的内容。与其他文档类似，簇中剩余的文档也包含 URL 及文本，如图 4-11 所示。

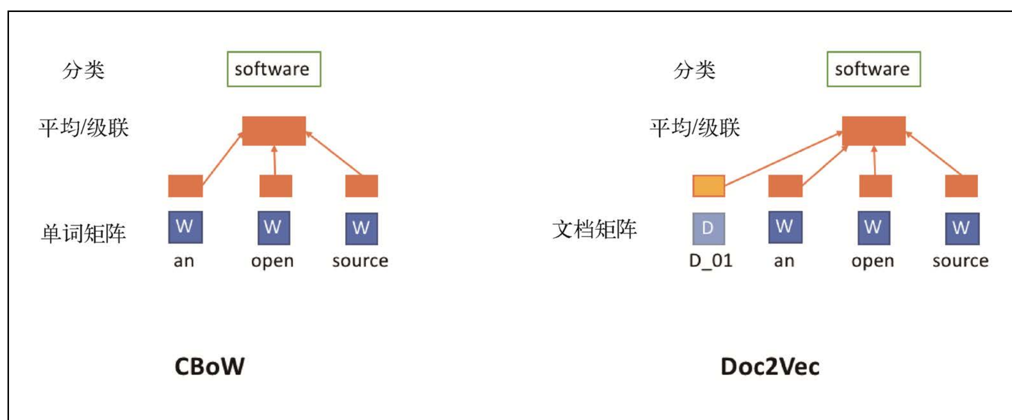



图 4-11

图 4-11 将 Doc2vec 与连续词袋的 Word2vec 进行了比较。这也强调了从 Word2vec 到 Doc2vec 模型的清晰拓展：从学习词向量到文档向量。

## 4.5 小结

本章讨论了 Word2vec 及其变体，介绍了用于理解词间关系的跳字模型代码，随后使用 TensorBoard 进行了词嵌入的可视化并查看了各种投影给可视化带来的帮助。接下来讨论了从 Word2vec 到创建文档表示的逻辑拓展，并使用 TF-IDF 进行了模型提升及优化。最后，我们探讨了 Doc2vec 及其变体以建立文档级别的向量表示，并展示了通过 TensorBoard 查看文档向量以发现文档的主题。

在下一章，我们将了解使用深度神经网络进行文本分类，还会介绍用于构建文本分类器的不同神经网络并分别讨论其架构的优缺点。

# 第 5 章 使用 LSTM 进行文本分类

文本分类指的是将自然语言或非结构化文本标记为预定义集中的类别。它的应用包括识别产品评论中的积极或消极情绪，对新闻文章进行分类，以及根据顾客在社会媒体上对产品的交流进行客户细分等。一个文本分类的真实示例是在 Gmail 中使用**机器学习**进行垃圾邮件自动检测。本章的主要目的是使你理解并熟悉用于文本分类的深度学习实战方法。

第 2 章简要提及了如何应用经典机器学习方法用 NLTK 和 sklearn 的词袋模型来进行文本分类。本章将为更深入地研究如何使用深度学习方法进行文本分类，并将重点放在使用 RNN（例如 LSTM 和 GRU）进行文本分类上。但为完整起见，我们还将介绍 CNN。为拓宽该话题，我们还将介绍一个相关的无监督学习方法，名为**主题建模**。总结一下，以下是本章涵盖的主题：

- 主题建模
- 使用 CNN 进行文本分类
- 使用 RNN 进行文本分类
- 基于迁移学习的文本分类
- 用于文本分类的最新深度学习方法概述

## 5.1 文本分类数据

在深入探讨文本分类中的机器学习问题之前，我们看一下网上可用的各种开放数据集。许多分类任务可能需要大批量标注的文本数据，而这些数据可大致分为二分类、多分类和多标签的[①]。表 5-1 展示了在研究和某些比赛（例如Kaggle）中用于基准测试的一些流行数据集。

表 5-1

| | 数据集名称 | 类别 |
|---|---|---|
| 1 | IMDb movie Dataset | 二分类 |
| 2 | Twiiter Sentiment Analysis Dataset | 二分类 |

① 多分类和多标签的区别在于，多分类数据仅属于同一类别，而多标签数据可以属于多个类别。——译者注

（续）

| | 数据集名称 | 类　别 |
|---|---|---|
| 3 | YouTube Spam Collection Dataset | 二分类 |
| 4 | News Aggregator Dataset | 多分类 |
| 5 | Yelp reviews | 多标签 |
| 6 | Amazon reviews | 多分类 |
| 7 | Reuters Corpora | 多标签/多分类 |

本章也会使用之前用过的一些数据集作为示例。尽管表 5-1 并不详尽，但其有助于你开始尝试文本分类和主题分类任务。

## 5.2　主题建模



当我们有一个文档集合却不清楚其中各个文档的分类时，主题模型可以帮助我们找到文档所属的大致类别。该模型将每个文档都视为主题的混合，其中可能包含一个主要主题。

比如，假设有如下句子：

- Eating fruits as snacks is a healthy habit
- Exercising regularly is an important part of a healthy lifestyle
- Grapefruit and oranges are citrus fruits

这些句子的主题模型输出可能会如下所示。

- **主题 A**：40% healthy、20% fruits、10% snacks
- **主题 B**：20% Grapefruit、20% oranges、10% citrus
- **句子 1 和 2**：80%主题 A、20%主题 B
- **句子 3**：100%主题 B

从模型的输出可以推测出主题 A 与健康（health）有关，主题 B 与水果（fruits）有关。尽管我们先前不知道这些主题，但是该模型输出了文档中与健康、运动和水果相关的单词的概率。

从这些例子可以清楚地看出，主题建模作为一种无监督的学习方法，在几乎没有用于文本分类的标签时，有助于发现文档的结构或模式。主题建模最流行的算法是**隐含狄利克雷分布**（Latent Dirichlet Allocation，LDA）。LDA 的原始论文使用变分贝叶斯方法来估计单词属于不同主题的概率。该算法的详细信息可以在论文“Latent Dirichlet Allocation”中找到。因为这超出了本书的范围，所以本书不作赘述。下面来看一个使用 LDA 进行主题建模的示例。对于 LDA 模型，我们将使用 gensim 库从 NLTK 网络文本语料库的示例文本中找到主题（第 2 章已对该语料库做了介绍）。该示例的完整 Jupyter Notebook 可在本书代码库中找到（Chapter05/01_example.ipynb）。首先，我们将为示例导入必要的 Python 模块：

```
from nltk.corpus import webtext, stopwords
import gensim
import random
from pprint import pprint
import numpy as np
import logging
#logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
level=logging.INFO)
```

这里导入了 `nltk`、`webtext` 和 `stopwords` 语料库。如需使用 gensim 模块，应使用 `pip` 安装器来安装：

```
pip install gensim
```

以上代码会在系统上安装 gensim，接下来的代码将从对应文本语料库中读入句子：

```
firefox = webtext.sents('firefox.txt')
wine = webtext.sents('wine.txt')
pirates = webtext.sents('pirates.txt')
```

`sents` 函数从各个文本语料库中读入句子。来自 Firefox 论坛的句子、电影《加勒比海盗》（*Pirates of the Caribbean*）的台本以及 `wine` 的评论都通过以下代码被添加到了列表中：

```
all_docs = []
all_docs.extend(firefox)
all_docs.extend(pirates)
all_docs.extend(wine)
random.shuffle(all_docs)
```

这将把所有文本文档整理进 Python 列表中，并对列表进行混洗以转换为随机文本列表集合。稍后我们将使用以下代码来验证主题模型能否区分来自 NLTK 网络文本语料库的三个不同主题类别的句子：

```
docs = [[word for word in doc if word not in stopwords.words('english')]
for doc in all_docs]
docs = [doc for doc in docs if len(doc)&gt;1]
```

在以下代码中，我们使用 NLTK 中的 `stopwords` 语料库从文本中移除停用词，同时忽略单词长度小于 `1` 的所有文本：

```
chunksize=len(docs)
dictionary = gensim.corpora.Dictionary(docs)
corpus = [dictionary.doc2bow(doc) for doc in docs]
model = gensim.models.LdaModel(corpus=corpus, id2word=dictionary,
num_topics=3,passes=20)
```

我们使用 gensim 中的 `dictionary` 类将文本集合 `docs` 转换为词袋表示并存储在 `corpus` 变量中。字典将文档转换为稀疏词袋向量，并且赋予每个单词或词元对应的 ID。生成的词袋向量与对应字典将被传递给 `LdaModel` 以训练 ID 到单词的映射。同样将主题数设置为 `3`，这也是

我们从 NLTK webtext 语料库中获得的主题数目。最后，将模型的 passes（训练的轮数）设置为 20。以下代码完成了从语料库提取热门主题的操作：

```
top_topics = model.top_topics(corpus)
print("Topic 1: ")
print(top_topics[0][0])
print("Topic 2: ")
print(top_topics[1][0])
print("Topic 3: ")
print(top_topics[2][0])
```

以下代码显示了这些主题以及各个单词在其中出现的概率：

```
Topic 1:
[(0.029538315, '.'), (0.025298702, '"'), (0.018974159, "'"), (0.017001661,
'-'), (0.0097839413, '('), (0.0089947991, 'page'), (0.0080595175, ')'),
(0.0076006982, 'window'), (0.0075753955, 'Firefox'), (0.0061700493,
'open'), (0.0058493023, 'menu'), (0.0057583884, 'bar'), (0.005752211, ':'),
(0.0057242708, 'tab'), (0.0054682544, 'new'), (0.0053855875, 'Firebird'),
(0.0052021407, 'work'), (0.0050605903, 'browser'), (0.00455163, '0'),
(0.0045419205, 'button')]

Topic 2:
[(0.10882618, '.'), (0.048713163, ','), (0.033278842, '-'), (0.019521466,
'I'), (0.018609792, '***'), (0.011298033, 'fruit'), (0.010273052, 'good'),
(0.0097078849, 'A'), (0.0089780623, 'wine'), (0.0089215562, "'"),
(0.0087491088, 'bit'), (0.0080983331, 'quite'), (0.0072782212, 'Top'),
(0.0061755609, '****'), (0.0060614017, '**'), (0.005842932, 'nose'),
(0.0057750815, 'touch'), (0.0049686432, 'Bare'), (0.0048470194, 'Very'),
(0.0047901836, 'palate')]

Topic 3:
[(0.051035155, ','), (0.043318823, ':'), (0.037644491, '.'), (0.029482145,
'['), (0.029230012, ']'), (0.023068342, "'"), (0.019555457, '!'),
(0.012494524, 'Jack'), (0.011483309, '?'), (0.010315109, '*'),
(0.008776715, 'JACK'), (0.008776715, 'SPARROW'), (0.0074223313, '-'),
(0.0061529884, 'WILL'), (0.0061529884, 'TURNER'), (0.0060977913, 'Will'),
(0.0055771996, 'I'), (0.0054870662, '...'), (0.0041205585, 'ELIZABETH'),
(0.0041205585, 'SWANN')]
```

尽管输出看上去甚至包含了标点符号，但基于单词在各个主题中的出现频率仍可以看出模型输出的一些模式。Topic 1 似乎与 Firefox 论坛有关，Topic 2 与 wine 评论相符，而 Topic 3 则来自于《加勒比海盗》电影台本。

## 主题建模与文本分类

正如以上示例所示，主题建模之间并非互斥的，这是因为它将文档建模为主题的混合。比如，可以将一个文档分类为 70%主题 A 及 30%主题 B。但文本分类则唯一地将文档归为特定类别。然而，文本分类需要经过标记的数据，而这些数据可能并不总是可用的。我们也可以将主题模型

的输出用作文本分类的特征，这在某种程度上可能会提高分类的准确率。主题模型还可用于快速创建手动标注的数据并随后将其用于训练文本分类器。我们已经学习了主题建模，并看到了它与文本分类的区别及联系。现在让我们使用深度学习模型探索文本分类吧。

## 5.3　用于文本分类的深度学习元架构

深度学习文本分类模型通常由如下三部分按顺序相连组成：

- 嵌入层
- 深层表示组件
- 全连接部分

我们将以下几小节中逐一进行讨论。

### 5.3.1　嵌入层

给定一系列单词 ID 作为输入，嵌入层会将这些 ID 转换为密集词向量的输出列表。正如我们在第 3 章所看到的那样，词向量能够捕获单词间的语义。在诸如 TensorFlow 的深度学习框架中，这部分工作通常由存储了查找表的嵌入查找层处理，将数字 ID 所表示的单词映射到密集向量表示形式。

### 5.3.2　深层表示

深层表示将嵌入向量序列作为输入，并将其转化为压缩后的表示形式。压缩表示能有效捕获文本中单词序列的所有信息。深度表示部分通常由 RNN 获取，但也可以使用 CNN 完成。对于 RNN，文本的压缩表示形式是最后一个时间步之后网络输出的最终隐藏状态。在 CNN 中，这则是最后一层的输出（通常是最大池化层）。因此，RNN 和 CNN 输出均能够捕获文本输入的深层表示。

### 5.3.3　全连接部分

**全连接部分**从 RNN 或 CNN 中获取深度表示，并将其转换为最终输出的类别或类别所对应的分数。该组件由全连接层、批标准化和用于正则化的 dropout 层（可选）组成。

现在，我们已经看到了深度学习文本分类器的通用元架构。在以下各节中，我们将结合该架构以深入研究文本分类器的实例。具体来说，我们将研究如何识别 YouTube 视频的垃圾评论以及如何对新闻文章进行分类。

## 5.4　使用 RNN 识别 YouTube 视频垃圾评论

作为首个示例，我们将研究在 YouTube 视频评论中识别垃圾评论的问题。该示例的完整 Jupyter Notebook 文件可在本书代码存储库中找到（Chapter05/02_example.ipynb）。所用数据包含带有二进制标签的评论，这些标签指明了评论是真实的还是垃圾。以下代码将 CSV 格式的评论加载到 pandas DataFrame 中：

```
comments_df_list = []
comments_file = ['data/Youtube01-Psy.csv','data/Youtube02-
KatyPerry.csv','data/Youtube03-LMFAO.csv',
 'data/Youtube04-Eminem.csv','data/Youtube05-Shakira.csv']
for f in comments_file:
 df = pd.read_csv(f,header=0)
 comments_df_list.append(df)
comments_df = pd.concat(comments_df_list)
comments_df = comments_df.sample(frac=1.0)
print(comments_df.shape)
comments_df.head(5)
```

5

如表 5-2 所示输出显示了不同领域的 YouTube 评论示例。

表　5-2

| | COMMENT_ID | AUTHOR | DATE | CONTENT | CLASS |
|---|---|---|---|---|---|
| 102 | z12dfr5irwr5chwm3232gvnq2laqcdezn04 | Carlos Rueda | 2015-05-22T15:04:20.310000 | I am going to blow my mind | 0 |
| 117 | z133ibkihkmaj3bfq22rilaxmp2yt54nb | Debora Favacho (Debora Sparkle) | 2015-05-21T14:08:41.338000 | BEST SONG EVER X3333333333 | 0 |
| 331 | _2viQ_Qnc68Qq98m0mmx4rlprYiD6aYgMb2x3bdupEM | Hidden Love | 2013-08-01T09:19:56.654000 | Hi. Check out and share our songs. | 1 |
| 322 | z13cedgolkfvw3xey22kcnzrfm3egjj0z | Rafael Diaz Jr | 2015-01-25T20:57:46.039000 | Check out this video on YouTube: | 1 |
| 133 | LneaDw26bFugQanw0UtVOqzEgWt6mBD0k6SsEV7u968 | Jacob Johnson | NaN | You guys should check out this EXTRAORDINARY w… | 1 |

这里，我们将来自五个受欢迎 YouTube 视频的.csv 文件加载到 pandas DataFrame 中。从输出中可以看到，它同时显示了有效评论和垃圾评论。CONTENT（内容）列包含评论的文本，而 CLASS（类别）列则将垃圾评论设置为 1、有效评论设置为 0。通过使用以下代码，我们将所有评论长度的平均值用作每个评论的最大长度，凡单词数大于最大限制的评论都将被截断以保持训练数据的大小不变：

```
average_comments_size = int(sum([len(c) for c in
comments_df.CONTENT])/comments_df.shape[0])
print(average_comments_size)
```

我们使用 vocabulary_processor 来预处理所有评论，并将数据划分为 80%的训练集和 20%的测试集，如以下代码所示：

```
vocabulary_processor =
tf.contrib.learn.preprocessing.VocabularyProcessor(average_comments_size)
X_transform = vocabulary_processor.fit_transform(comments_df.CONTENT)
X_transform = np.array(list(X_transform))
y = comments_df.CLASS.values
X_train, X_test, y_train, y_test =
model_selection.train_test_split(X_transform,
 y, test_size=0.2, random_state=42)
n_words = len(vocabulary_processor.vocabulary_)
```

在本例以及本章后续示例中，我们将利用 TensorFlow estimator API 来创建、训练和测试模型。TensorFlow 中的评估器（estimator）提供了一个易于使用的界面，可用于构建图形、初始化变量、创建检查点文件并保存摘要供 TensorBoard 查看。使用以下代码创建 estimator：

```
def get_estimator_spec(input_logits, out_lb, train_predict_m):
    preds_cls = tf.argmax(input_logits, 1)
    if train_predict_m == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(
        mode=train_predict_m,
        predictions={
            'pred_class': preds_cls,
            'pred_prob': tf.nn.softmax(input_logits)
        })
    tr_l = tf.losses.sparse_softmax_cross_entropy(labels=out_lb,
logits=input_logits)
    if train_predict_m == tf.estimator.ModeKeys.TRAIN:
        adm_opt = tf.train.AdamOptimizer(learning_rate=0.01)
        tr_op = adm_opt.minimize(tr_l,
global_step=tf.train.get_global_step())
        return tf.estimator.EstimatorSpec(train_predict_m, loss=tr_l,
train_op=tr_op)
    eval_metric_ops = {'accuracy': tf.metrics.accuracy(labels=out_lb,
predictions=preds_cls)}
    return tf.estimator.EstimatorSpec(train_predict_m, loss=tr_l,
train_op=tr_op)
```

我们使用 AdamOptimizer 来优化 tf.losses.sparse_softmax_cross_entropy 损失函数。当给定 logit 模型及两个类的概率分布和真实标签时，它能计算交叉熵：

```
def rnn_model_fn(features, labels, mode):
    comments_wd_vec = tf.contrib.layers.embed_sequence(
      features[COMMENTS_FT], vocab_size=n_words, embed_dim=EMBED_DIMENSION)
    comments_word_list = tf.unstack(comments_wd_vec, axis=1)
    rnn_cell = tf.nn.rnn_cell.GRUCell(average_comments_size)
    _, comments_encoding = tf.nn.static_rnn(rnn_cell, comments_word_list,
dtype=tf.float32)
    logits = tf.layers.dense(inputs=comments_encoding, units=2,
activation=None)
```

```
    return get_estimator_spec(input_logits=logits, out_lb=labels,
train_predict_m=mode)
```

如上一节所述，在用于模型文本分类的元架构中，我们使用了一个嵌入层，后接一个 GRUCell。GRU 的输出被馈送到计算 logits 的密集层，然后 logits 的输出被传递到 softmax 层以计算各个类别的预测概率。本例所使用的 GRU 单元与 LSTM 相似，不同之处在于前者输出的隐藏状态不包含控制门（因此与 GRU 相比，LSTM 多了一个额外的门）。除此之外，GRU 还可能也无法记住长期的单词联系。但是对于此特定任务，两者的差异并不明显。你还可以尝试在代码中使用 LSTM 去替换 GRU 单元，具体代码如下所示：

```
run_config = tf.contrib.learn.RunConfig()
run_config =
run_config.replace(model_dir='/tmp/models/',save_summary_steps=10,log_step_
count_steps=10)
classifier =
tf.estimator.Estimator(model_fn=rnn_model_fn,config=run_config)
```

5

我们使用 RunConfig 在/tmp/models 目录下保存模型检查点，并且将日志记录和摘要步进频率从默认值 100 修改为 10，以便更好地在 TensorBoard 中显示。你也可以在代码中对于这些值进行相应修改。最后，使用以下代码对模型进行 200 步训练：

```
train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={COMMENTS_FT: X_train},
      y=y_train,
      batch_size=128,
      num_epochs=None,
      shuffle=True)
classifier.train(input_fn=train_input_fn, steps=200)
Output
INFO:tensorflow:Saving checkpoints for 200 into /tmp/models/model.ckpt.
INFO:tensorflow:Loss for final step: 0.000836024.
```

使用测试数据对模型进行评估，其准确率为 94%，如以下代码所示：

```
test_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={COMMENTS_FT: X_test},
      y=y_test,
      num_epochs=1,
      shuffle=False)
preds = classifier.predict(input_fn=test_input_fn)
y_predicted = np.array(list(p['pred_class'] for p in preds))
y_predicted = y_predicted.reshape(np.array(y_test).shape)

acc = metrics.accuracy_score(y_test, y_predicted)
print('Accuracy: {0:f}'.format(acc))
```

以下输出显示了先前代码的结果：

```
INFO:tensorflow:Restoring parameters from /tmp/models/model.ckpt-200
Accuracy: 0.905612
```

要可视化图和训练过程，需要启动 TensorBoard，且 `logdir` 值指向在 `RunConfig` 中所使用的相同路径。使用浏览器访问 localhost:6006（默认值），查看图及绘图结果。可以看到，损失随着训练步骤而稳定减少，如图 5-1 所示。

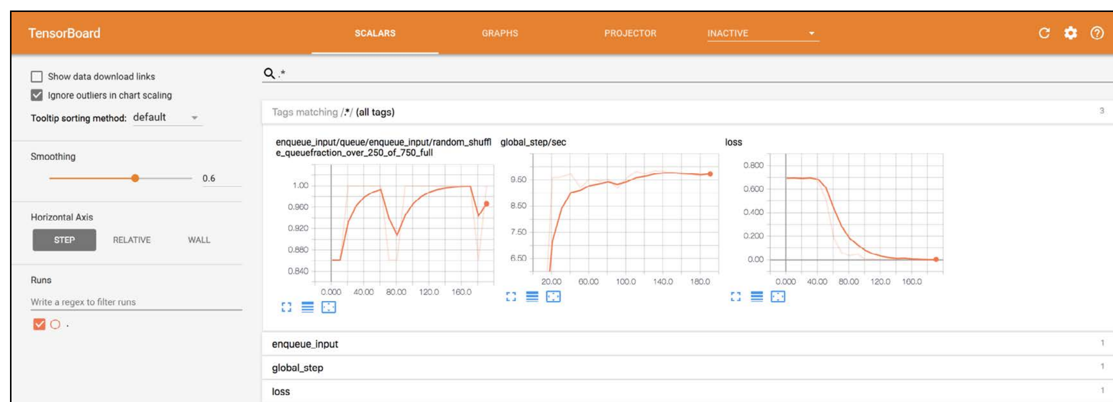



图 5-1　训练损失及步数/秒

模型图的可视化结果展示了输入、嵌入层、RNN 单元、密集层和 softmax 输出，如图 5-2 所示。

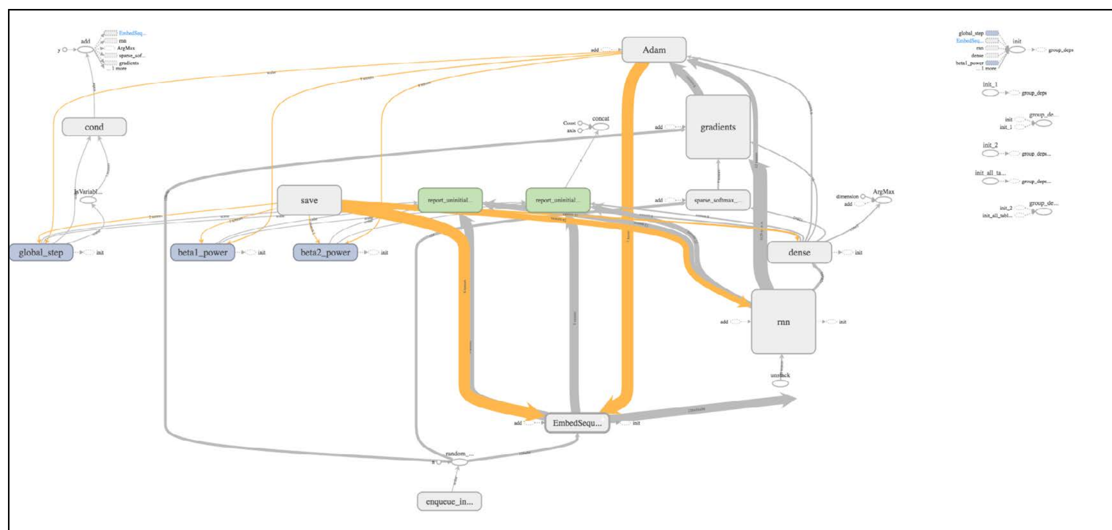



图 5-2　文本分类模型

通过嵌入投影还可以看出，模型学习到的词嵌入被明显地分成了两个簇。图 5-3 显示了垃圾评论相关单词和真实评论相关单词的示例。这显示了模型是如何将与这两个类相关联的词向量压入单独的簇中的。

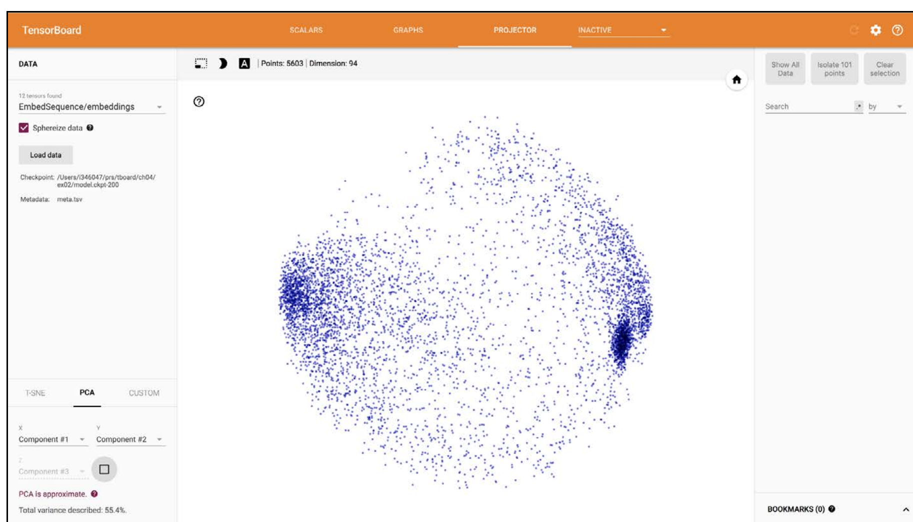



图 5-3　经 PCA 后的可视化词向量

我们看到了如何使用基于 RNN 的深度学习模型进行文本分类。如先前所述，我们还可以使用 CNN 进行该工作，接下来就对此进行探讨。

## 5.5　使用 CNN 对新闻主题分类

在本例中，我们将使用新闻聚合器所收集的新闻网页的数据集。数据集中有四个类别，分别属于科技新闻、商业新闻、娱乐新闻和健康新闻。该示例的完整 Jupyter Notebook 文件可在本书代码库中找到（Chapter05/03_example.ipynb）。

首先看看数据集中的数据示例：

```
news_df = pd.read_csv('data/newsCorpora.csv',delimiter='\t', header=None,
names=['ID','TITLE','URL','PUBLISHER','CATEGORY','STORY','HOSTNAME','TIMEST AMP'])
news_df = news_df.sample(frac=1.0)
news_df.head(5)
```

数据集则表示为如表 5-3 所示的形式。

表　5-3

| ID | TITLE | CATEGORY |
| --- | --- | --- |
| 225897 | Fed's Dudley sees *relatively slow* rate hike... | b |
| 26839 | L'Wren Scott's death officially ruled as suicide | e |
| 32332 | Idina Menzel Says She *Benefited* From John Tr... | e |
| 188058 | Weak earnings and the dark cloud of Ukraine co... | b |
| 22176 | Android Wear-Google's latest PLAY | t |

此处，我们只对标题（TITLE）和类别（CATEGORY）列感兴趣。类别列被设置为新闻所属的四个类别之一。这四个类别是科技、商业、娱乐和健康，分别由 t、b、e 和 m 表示。与前面的示例一样，我们将标题的平均大小作为最大阈值来固定训练实例的长度。

TensorFlow 中的 `vocabulary_processor` 将每个标题文本预处理为单词列表，列表的长度固定为平均标题大小。单词数如果大于平均标题大小，则文本将被截断；如果小于平均标题大小，则用零填充。这可以通过以下代码实现：

```
lencoder = LabelEncoder()
voc_processor =
tf.contrib.learn.preprocessing.VocabularyProcessor(average_title_size)
X_transform = voc_processor.fit_transform(news_df.TITLE)
X_transform = np.array(list(X_transform))
y = lencoder.fit_transform(news_df.CATEGORY.values)
X_train, X_test, y_train, y_test =
model_selection.train_test_split(X_transform,
                                    y, test_size=0.2, random_state=42)
n_words = len(voc_processor.vocabulary_)
n_classes = len(lencoder.classes_)
```

这里依旧使用带有稀疏 softmax 交叉熵的 `AdamOptimizer` 来训练模型。对于该模型，我们将前一个示例中的 `GRUCell` 替换为卷积层，然后进行最大池化，其中过滤器的数量设置为 5、高度设置为 3、宽度设置为嵌入尺寸。按照步幅参数的设置，单个步幅执行三个单词的滤波器卷积。卷积层的数量被设置为一层，以简化模型架构并加快训练速度。你也可以尝试增加过滤器和卷积层的数量。需要注意的是，我们还将嵌入输入转为四个张量且最后一个通道的维度为 1。以下代码显示了 CNN 模型的构造过程：

```
filter_size=3
num_filters=5
def cnn_model_fn(features,labels,mode):
    news_word_vectors = tf.contrib.layers.embed_sequence(features[NEWS_FT],
vocab_size=n_words,
embed_dim=WORD_EMBEDDING_SIZE)
    news_word_vectors = tf.expand_dims(news_word_vectors, -1)
    filter_shape = [filter_size, WORD_EMBEDDING_SIZE, 1, num_filters]
    W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1),
name="W")
    b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
    conv1 = tf.nn.conv2d(news_word_vectors,
            W,
            strides=[1, 1, 1, 1],
            padding="VALID",
            name="conv1")
    relu1 = tf.nn.relu(tf.nn.bias_add(conv1, b), name="relu")
    pool1 = tf.nn.max_pool(
            relu1,
            ksize=[1, average_title_size - 3 + 1, 1, 1],
            strides=[1, 1, 1, 1],
            padding='VALID',
            name="pool1")
```

```
    activations1 = tf.contrib.layers.flatten(pool1)
    logits =
tf.contrib.layers.fully_connected(activations1,n_classes,activation_fn=None
)
    return get_estimator_spec(input_logits=logits, out_lb=labels,
train_predict_m=mode)
```

我们仍对模型进行 200 步训练并使用测试数据展开评估，如以下代码所示：

```
run_config = tf.contrib.learn.RunConfig()
run_config =
run_config.replace(model_dir='/tmp/models/',save_summary_steps=10,log_step_
count_steps=10)
classifier =
tf.estimator.Estimator(model_fn=cnn_model_fn,config=run_config)
train_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={NEWS_FT: X_train},
      y=y_train,
      batch_size=len(X_train),
      num_epochs=None,
      shuffle=True)
classifier.train(input_fn=train_input_fn, steps=100)
```

以下是测试数据输出的结果：

```
INFO:tensorflow:Restoring parameters from /tmp/models/model.ckpt-27
Accuracy: 0.903094
[[20990   534   108  1559]
 [  410 29606   142   331]
 [  493  1432 5741  1381]
 [ 1266   369   162 19960]]
```

我们可以看到准确率和混淆矩阵：准确率约为 93%，而混淆矩阵的对角线列则显示了正确预测的文档类型。我们可以在 TensorBoard 中可视化模型图和学习指标图，如图 5-4 所示。

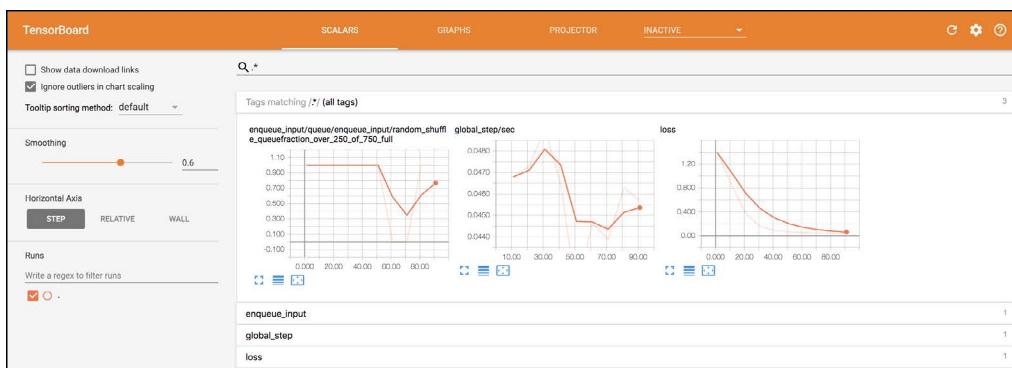



图 5-4　训练损失及步数/秒

随着训练步数的增多，训练损失不断下降。从图 5-4 中可以看出，当增加卷积层和过滤器的数量时，准确率可以得到改善。

图 5-5 显示了输入、嵌入层、卷积层、最大池化、密集层和 softmax 输出。

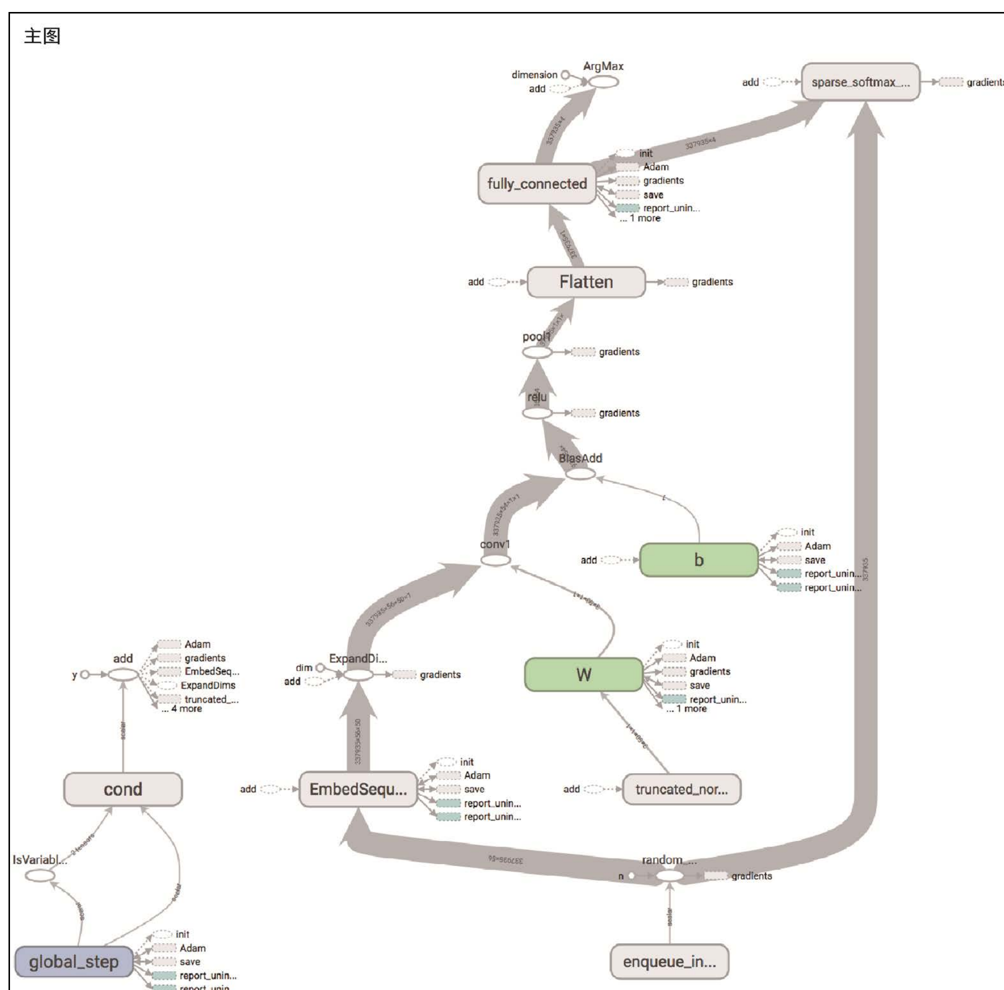



图 5-5　用于文本分类的 CNN 模型

我们已经看到了如何将相同的神经网络元架构用于文本分类。这两个示例之间唯一的区别在于，第一个示例使用 RNN 作为深度表示，而第二个则使用了 CNN。但在两个示例中，输入的词嵌入都是使用训练数据从头开始学习的，而在训练数据较少的情况下，我们可以转向其他方法，如迁移学习。在下一个示例中，我们将使用预学习好的词嵌入进行文本分类。

## 5.6　使用 GloVe 嵌入进行迁移学习

**全局向量**（global vector，GloVe）使用大型文本语料库中的单词全局共现统计得出单词的密

集向量表示形式。这是一种无监督的学习方法，目的在于使学习到的向量的点积等于共现概率的对数。嵌入空间中向量之差将被转换为因为比率之比的对数，即比率取对数后的差值。

本例将使用在 Twitter 数据上预先训练好的 GloVe 嵌入。该数据包含约 20 亿条推文且词汇量大小为 120 万。对于分类任务，我们将采用用户对于 Amazon 即时短视频的评论或评分。首先，以 JSON 格式加载评论数据，并将其转换为 pandas DataFrame 格式，如以下代码所示：

```
json_data = []
with gzip.open('data/reviews_Amazon_Instant_Video_5.json.gz', 'rb') as
json_file:
for json_str in json_file:
 json_data.append(json.loads(json_str))
 reviews_df = pd.DataFrame.from_records(json_data)
```

对于分类任务，我们只对 `reviewText` 栏和 `overall` 栏感兴趣，因为它们分别代表了用户的评论和评分。使用如下代码列出示例数据，并对评分和评论文本进行查看。

```
reviews_df[['overall','reviewText']].head(5)
```

下表是先前代码的输出结果。

| | overall | reviewText |
|---|---|---|
| 9102 | 4.0 | I enjoy this show because it shows old stuff t... |
| 22551 | 5.0 | I really enjoy these programs. I am a pretty g... |
| 33811 | 3.0 | Decent cast, but it seems a little formulaic... |
| 8127 | 5.0 | My kids love this show. It’s one of my 3 year... |
| 17544 | 5.0 | How they keep coming up with the shenanigans t... |

接着使用如下代码从 GloVe 嵌入文本文件中载入词向量：

```
def build_word_vector_matrix(vector_file):
    np_arrays = []
    labels_array = []
    with codecs.open(vector_file, 'r', 'utf-8') as f:
        for i, line in enumerate(f):
            sr = line.split()
            if(len(sr)&lt;26):
                continue
            labels_array.append(sr[0])
            np_arrays.append(np.array([float(j) for j in sr[1:]]))
    return np.array(np_arrays), labels_array
```

文件中的每一行包含单词及其相应向量表示形式。将其读入 NumPy 数组 `np_arrays`，并将相应的单词读入 `labels_array`。调用 `build_word_vector_matrix` 时，将返回单词词元和对应词向量数组。我们依旧使用 TensorFlow 的 `vocabulary_processor` 将评论数据转换为固定长度的句子，且该长度等于所有评论的平均大小。将 DataFrame 评论传递给 `vocabulary_processor` 来实现这一点，如以下代码所示：

```
lencoder = LabelEncoder()
voc_processor =
tf.contrib.learn.preprocessing.VocabularyProcessor(average_review_size)
voc_processor.fit(vocabulary)
X_transform = voc_processor.transform(reviews_df.reviewText)
X_transform = np.array(list(X_transform))
y = lencoder.fit_transform(reviews_df.overall.values)
X_train, X_test, y_train, y_test =
model_selection.train_test_split(X_transform,
                                    y, test_size=0.2, random_state=42)
n_words = len(voc_processor.vocabulary_)
n_classes = len(lencoder.classes_)
```

评估器（estimator）的模型函数依旧使用了 RNN，但我们为输入的嵌入变量 word_embeddings 设置了 Trainable = False 选项，确保模型在训练期间不会再次学习嵌入。请注意，word_embeddings 变量包含 GloVe 嵌入的查找表。以下代码显示了用于创建 RNN 模型的函数：

```
def rnn_model_fn(features,labels,mode):
    em_plholder = tf.placeholder(tf.float32, [voc_size, WD_EMB_SIZE])
    Wt = tf.Variable(em_plholder,trainable=False, name='Wt')
    comments_word_vec = tf.nn.embedding_lookup(Wt, features[REVIEW_FT])
    comments_wd_l = tf.unstack(comments_word_vec, axis=1)
    rnn_cell = tf.nn.rnn_cell.GRUCell(WD_EMB_SIZE)
    _, comments_encoding = tf.nn.static_rnn(rnn_cell, comments_wd_l,
dtype=tf.float32)
    dense = tf.layers.dense(comments_encoding, units=512,
activation=tf.nn.relu)
    dropout = tf.layers.dropout(inputs=dense,
rate=0.4,training=(mode==tf.estimator.ModeKeys.TRAIN))
    logits = tf.layers.dense(inputs=dropout, units=n_classes)
    return get_estimator_spec(input_logits=logits, out_lb=labels,
train_predict_m=mode,
embedding_placeholder=em_plholder)
```

正如我们先前所做的那样，可以在 TensorBoard 里查看模型图和训练过程。可以观察到，当训练步数增加时，学习率（learning rate）稳定减小，如图 5-6 所示。

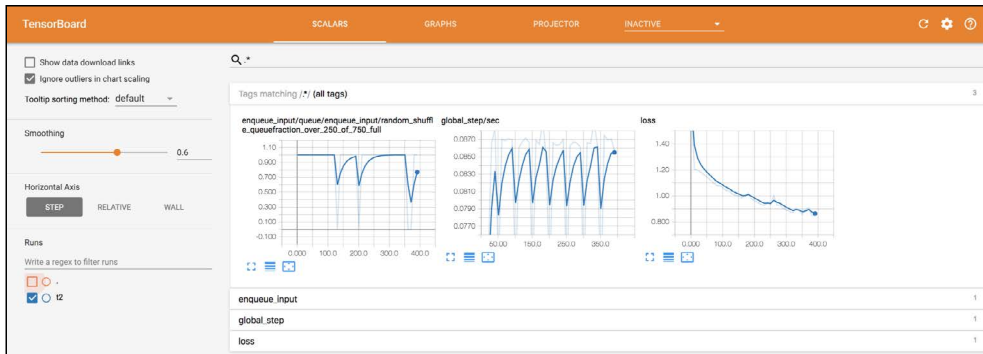



图 5-6　训练损失及步数/秒

图 5-7 显示了包含 `word_embedding` 输入、RNN 层、全连接（FC）层和最终 softmax 输出的网络。

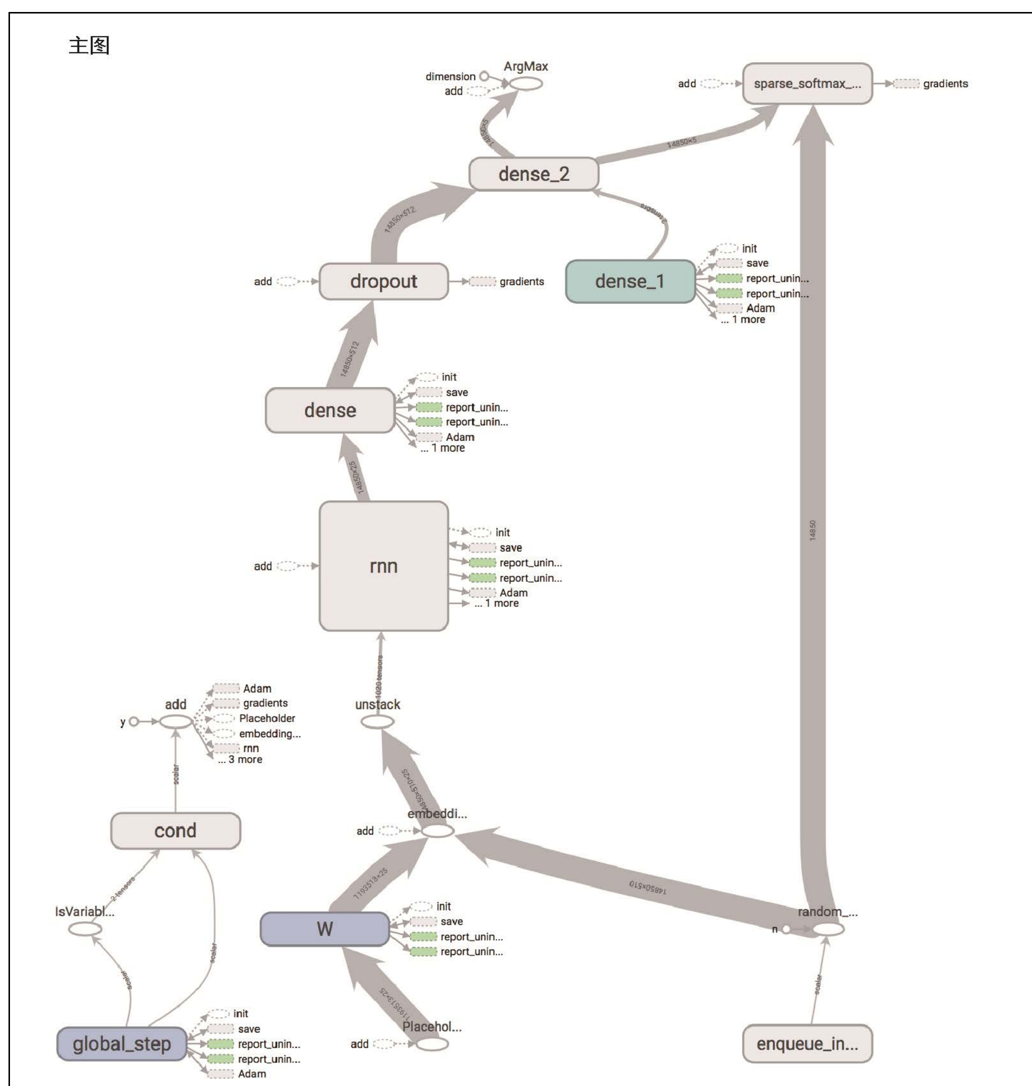



图 5-7　使用 GloVe 嵌入的模型

## 5.7　多标签分类

到目前为止，我们已经看到了需要将文本分类为某一类或者标签的问题。在文本分类问题中，可能要为同一个文档赋予多个类别。例如在 Yelp 评论数据集中，评论者可能在谈论餐厅的不同

方面，例如食物、氛围和服务质量等。在这种情况下，我们需要确定评论所属的类别，以便对整体评价有所了解。现在，我们将研究用来解决该问题的一些现有方法。

### 5.7.1　二元关联

用于标识文档 $L$ 个标签的多标签分类可以转换为 $L$ 个二进制分类问题。在该方法中，我们将文档传递到 $L$ 个二进制分类器中，其中每个分类器都经过训练以识别对应分类。$L$ 个分类器的输出将被合并以生成文档所属的类标签向量。即使如决策树之类的简单模型，也可以使用 SVM 来进行二进制分类。

该方法尽管最为简单，但缺点在于它假定各个类别彼此独立（但情况并非总是如此）。例如在 Yelp 评论示例中，（餐厅的）氛围和服务质量类别可能相互关联。现在我们将探讨其他处理方法。

### 5.7.2　用于多标签分类的深度学习

在“Large-scale Multi-label Text Classification – Revisiting Neural Networks”一文中，Nam 等人通过使用带隐藏层的深层**多层感知器**（multi-layer perceptron，MLP）和产生对应标签分数的输出单元来解决此问题。他们使用标签预测器，基于秩损失函数的阈值将标签分数从深层网络转换为二分类。这种方法的细节可以在该论文中找到。

图 5-8 对该方法进行了阐述。

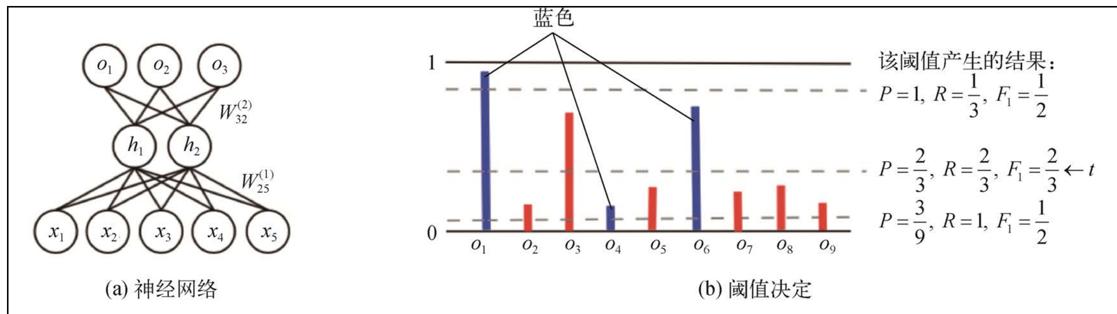



图 5-8　多标签分类方法

在图 5-8 中，输出由九个可能的标签组成，而且我们在这些标签上分别应用了阈值。选择产生最高F1 分数的阈值（正中间的那个）。蓝色[①]长条则是训练示例的相关标签。

有许多针对多类别问题的最新深度学习文本分类方法也可以输出针对多标签前 $k$ 个所属类预测（我们也可以应用阈值）。我们将研究其中一些可用于多标签分类的多分类模型。fastText

① 在纸质版图书中为深灰色。——编者注

是一种最近流行的深度学习方法。正如 A Joulin 等人在论文“Bag of Tricks for Efficient Text Classification”中所探讨的那样，fastText 通过平均化出现在文档中的词嵌入向量来表示文档。然后该平均文档向量将被传递到 softmax 层以输出所属类概率。因此，这种方法不考虑单词的顺序。在论文“Convolutional Neural Networks for Sentence Classification”中，Kim 等人使用 CNN 来串联的文档的词嵌入。他们在文档上使用多个过滤器，其输出被馈送到最大池化层，随后经过一个全连接层，最终由 softmax 输出对应于 $L$ 个标签的概率。这也是我们在示例中使用 CNN 进行文本分类的方法。在论文“Deep Learning for Extreme Multi-label Text Classification”中，Liu 等人利用 XML-CNN 架构进行多标签分类。特别要指出的是，之前所述的方法可能不适用于大量标签和/或标签的偏斜分布，而 XML-CNN 架构则试图取解决此问题。

图 5-9 对 XML-CNN 架构进行了阐述。

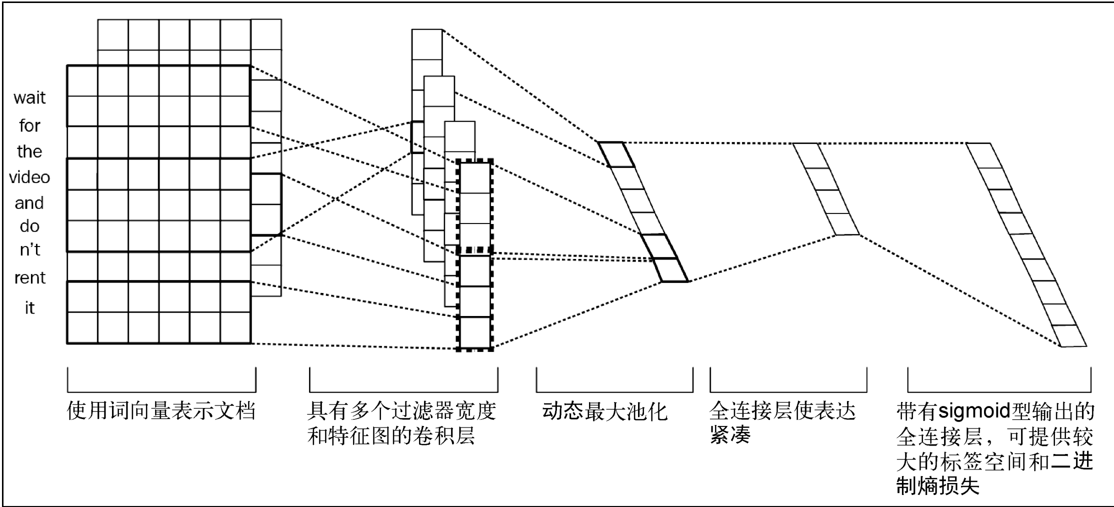



图 5-9

在图 5-9 中，该模型使用了具有动态最大池化的 CNN 和用于较大标签空间的全连接 sigmoid 型输出。基于 attention（注意力）的网络结构则是文本分类中的另一种最新技术，与迄今为止我们所描述的所有方法相比，它显示了出令人欢欣鼓舞的前景。下面，我们将查看一篇关于该主题的新论文。

### 5.7.3 用于文档分类的attention网络

在论文“Hierarchical Attention Networks for Document Classification”中，Yang 等人使用一种分层深度学习架构对文档进行分类。他们的灵感来源于文档具有固有的层次结构，例如单词构成句子、句子构成文档。此外，在捕获回答特定查询所需的语义和含义方面，文档中的所有部分并

非同样重要。他们使用了两种类型的 attention，一种在单词级别，而另一种在句子级别。除此之外，他们还将文本表示分为句子级别和文档级别。图 5-10 显示了该网络的不同组件，用于捕获单词和句子级别的深层表示以及相应的 attention 机制。

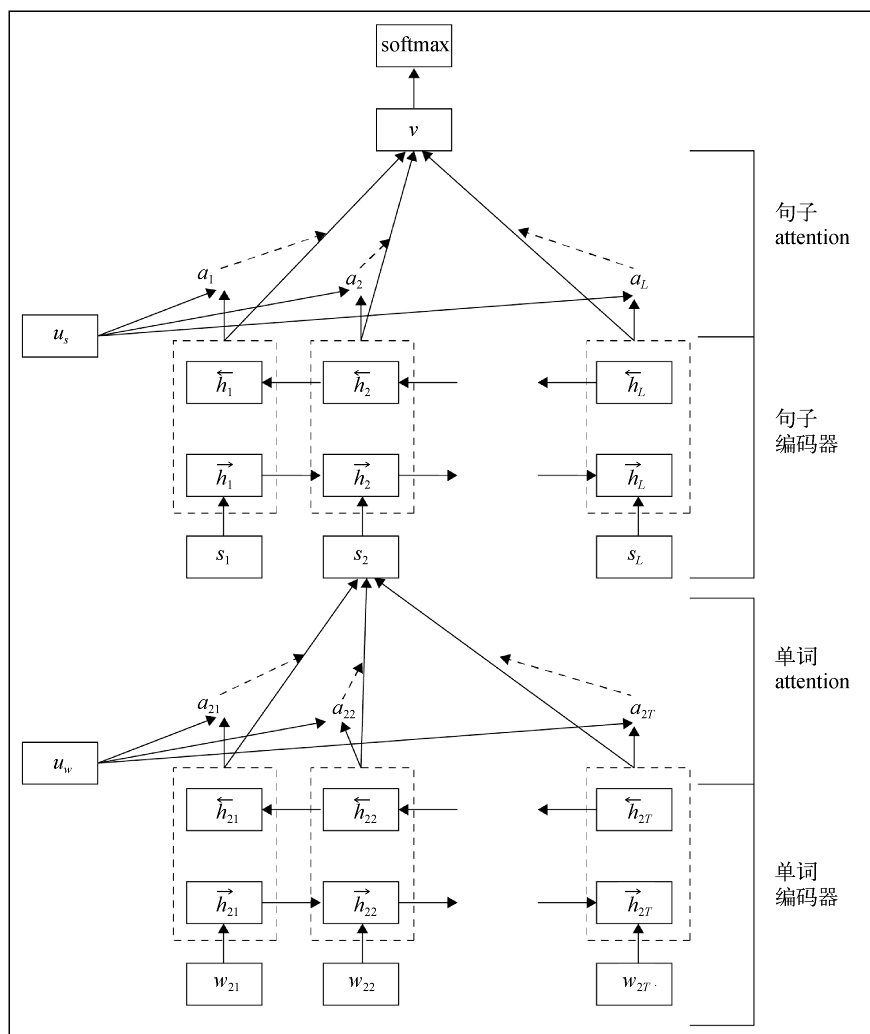



图 5-10　attention 网络组件

GitHub 中的项目和 ematvey/hierarchical-attention-networks 分别提供了本文所介绍模型的 Keras 实现和 TensorFlow 实现。

## 5.8　小结

本章探讨了使用 LSTM、GRU 和基于 CNN 的网络进行文本分类的不同技术，阐释了主题建模（文本分类中的一个相关问题），并介绍了一个使用 gensim 进行主题建模的简单示例。我们看到了一些使用真实数据集进行情感分类、评论评分预测和垃圾评论检测的解决方案。本章概述了如何使用 RNN 和 CNN 通过深度学习技术解决文本分类问题，还介绍了一个使用预训练词嵌入的迁移学习示例。最后，我们讨论了可用于复杂文本分类方案的最新技术，例如极端多标签分类和 attention 网络。

在下一章，我们将学习用于搜索和相似文档去重的深度学习方法。

# 第 6 章 使用 CNN 进行搜索和去重

事实证明，当提供大量数据点时，深度神经网络可以做得很好。但对于大多数引擎来说，主要问题在于缺乏用于构建大型搜索引擎的数据。搜索文本数据的传统方法涉及领域理解和关键字映射。这些为搜索引擎提供了包括相关主题足够信息的知识图谱，从而使搜索引擎能够找到答案。通过获取主题之间的关系，搜索引擎还能够拓展到新的主题上。

本章将探讨如何使用**机器学习**构建搜索引擎，并将其用于匹配和去重等任务。为了解深度学习是如何改进搜索的，我们将使用传统方法（例如 TF-IDF 和潜在语义索引）构建基准方法，并随后开发一个 CNN 以学习识别重复文本。我们将比较 CNN 与传统方法，以了解每种方法的利弊。

对于搜索任务，我们重视理解句子语义的能力，因为在大量数据中搜索文本需要对文本进行恰当的表示。我们将解决删除重复数据的问题来作为分类器的扩展。

## 6.1 数据

为了构建搜索和检索系统，我们将使用相同的数据进行训练和测试。我们将使用 Quora 重复问题数据来构建搜索和检索方法。Quora 重复问题对要处理的任务是确定两对问题是否具有相同的含义。该数据包含成对的问题和由人类专家标记的、关于该问题是否重复的真值标签。所提供的真值数据还提到这些标签是主观的，意味着并非所有人类专家都会就该问题达成共识。因此，由于文本数据固有的主观性，应将数据视为**有启示性的**，而非 100%准确。

### 数据描述

所提供的数据采用 `id`、`qid1`、`qid2`、`question1`、`question2` 和 `is_duplicate` 的格式，其中 `id` 字段提供训练对的 ID，`qid` 和 `qid2` 提供每个问题的 ID，而 `question1` 和 `question2` 则是用于训练的完整问题文本。`is_duplicate` 是布尔值、也是目标值，当文本对是重复的（语义上相同）时，其值设置为 1，否则设置为 0。训练数据包含大约 404 000 个问题对及其标签。

## 6.2 模型训练

接下来，我们训练模型以匹配这些问题对。导入相关库的代码如下所示：

```
import sys
import os
import pandas as pd
import numpy as np
import string
import tensorflow as tf
```

以下函数将 pandas 文本序列作为输入并将该序列转换为列表。列表中的每个项目都将被转换为小写字符串并去除周围的空白。整个列表最终被转换为 NumPy 数组并返回：

```
def read_x(x):
    x = np.array([list(str(line).lower().strip()) for line in x.tolist()])
    return x
```

以下函数则将 pandas 序列作为输入，转换为列表，最终转为 NumPy 数组并返回：

```
def read_y(y):
    return np.asarray(y.tolist())
```



以下函数拆分数据以进行训练和验证。验证数据有助于了解根据训练数据训练的模型对未知数据的泛化能力。通过对数据索引进行混洗，可以随机选择要验证的数据。该函数将问题对、问题对对应的标签以及拆分比作为输入：

```
def split_train_val(x1, x2, y, ratio=0.1):
    indicies = np.arange(x1.shape[0])
    np.random.shuffle(indicies)
    num_train = int(x1.shape[0]*(1-ratio))
    train_indicies = indicies[:num_train]
    val_indicies = indicies[num_train:]
```

此处将训练集和验证集的比设置为 10%。因此，通过对数组进行切片，可以对混洗后的索引分别进行训练和验证。由于索引已经被重新排序，可以将其用于拆分训练数据。输入数据有两组问题，即 `x1` 和 `x2`，并带有 `y` 标签以指示该对是否重复：

```
train_x1 = x1[train_indicies, :]
train_x2 = x2[train_indicies, :]
train_y = y[train_indicies]
```

类似地，验证数据被基于 10%的索引划分比率切片：

```
val_x1 = x1[val_indicies, :]
val_x2 = x2[val_indicies, :]
val_y = y[val_indicies]

return train_x1, train_x2, train_y, val_x1, val_x2, val_y
```

训练数据和验证数据都是从混洗后的索引选取的，数据也据此进行了切分。



注意，训练和验证问题对都应从索引中选取。

### 6.2.1　文本编码

以下函数能够将字符串列表转化为向量。该函数首先通过串联英文字符得到字符集合，接着将这些字符作为键生成字典，并将整数作为其键值。

```
def get_encoded_x(train_x1, train_x2, test_x1, test_x2):
    chars = string.ascii_lowercase + '? ()=+-_~"`<>,./\|[]{}!@#$%^&*:;' +
"'"
```

该示例仅仅是英文中的字符集，我们也可以纳入其他语言字符，以使该方法通用于不同的语言。



注意，该字符集可从数据集中得到，从而包含非英文字符。

下面将形成字符集与整数集的字符映射，以字典的形式供我们调用，如以下代码片段所示：

```
char_map = dict(zip(list(chars), range(len(chars))))
```

之后可以通过遍历所有问题获得句子的最大长度。首先生成包含每行长度的列表，并从其中获得所需的最大值：

```
max_sent_len = max([len(line) for line in np.concatenate((train_x1,
  train_x2, test_x1, test_x2))])
print('max sentence length: {}'.format(max_sent_len))
```

我们需要预设所有问题的最大长度以将向量量化为该固定大小。每当问题的长度小于预设大小时，只需在文本后补上空格：

```
def quantize(line):
       line_padding = line + [' '] * (max_sent_len - len(line))
       encode = [char_map[char] if char in char_map.keys() else char_map['
'] for char in line_padding]
        return encode
```

通过调用先前的 `quantize` 函数并将问题对转换为 NumPy 数组，可以实现对训练数据和测试数据的编码。每个问题都经过了迭代量化，如下所示：

```
train_x1_encoded = np.array([quantize(line) for line in train_x1])
train_x2_encoded = np.array([quantize(line) for line in train_x2])
test_x1_encoded = np.array([quantize(line) for line in test_x1])
test_x2_encoded = np.array([quantize(line) for line in test_x2])
return train_x1_encoded, train_x2_encoded, test_x1_encoded,
  test_x2_encoded, max_sent_len, char_map
```

接下来进行量化：在填充空格后，使用字符映射表对每个字符进行拆分和编码，然后将整数数组转换为 NumPy 数组。以下函数结合了前面的功能来实现数据的预处理：读取.csv 格式的数据以进行训练和测试。问题 1 和问题 2 来自数据帧的不同列，并被相应地拆分。无论问题对是否重复，数据都是二进制的。

首先，使用 pandas 框架载入包含训练数据集和测试数据集的.csv 文件：

```
def pre_process():
    train_data = pd.read_csv('train.csv')
    test_data = pd.read_csv('test.csv')
```

然后，将 pandas DataFrame 传递给开头所定义的函数，以便将原始文本隐式地转换为 NumPy 数组，如下所示。pandas 问题序列是被传递给函数之前的序列的子集：

```
train_x1 = read_x(train_data['question1'])
train_x2 = read_x(train_data['question2'])
train_y = read_y(train_data['is_duplicate'])
```

此处使用相同的函数将测试数据对转换为 NumPy 数组：

```
test_x1 = read_x(test_data['question1'])
test_x2 = read_x(test_data['question2'])
```

接下来，馈送训练数据和测试数据对应的 NumPy 数组以进行编码，并获取编码后的问题对、最大句子长度和字符映射表：

```
    train_x1, train_x2, test_x1, test_x2, max_sent_len, char_map = 
get_encoded_x(train_x1, train_x2, test_x1, test_x2)
    train_x1, train_x2, train_y, val_x1, val_x2, val_y = 
split_train_val(train_x1, train_x2, train_y)

    return train_x1, train_x2, train_y, val_x1, val_x2, val_y, test_x1, 
test_x2, max_sent_len, char_map
```

在训练过程中，也可以调用编码函数。

## 6.2.2　建立CNN模型

接下来，我们构建基于字符的 CNN 模型。我们从创建嵌入查找表开始，其大小为字符的数量（初始值为 50）。首先，从字符映射表获取字符集长度：

```
def character_CNN(tf_char_map, char_map, char_embed_dim=50):
    char_set_len = len(char_map.keys())
```

对于各个卷积层，使用随机值对其权重和偏差进行初始化。当模型建立完成后，卷积层函数将会被调用：

```
def conv2d(x, W, b, strides=1):
        x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1],
```

```
            padding="SAME")
        x = tf.nn.bias_add(x, b)
        return tf.nn.relu(x)
```

各个卷积层之后也都加上了最大池化层，其定义如下所示：

```
def maxpool2d(x, k=2):
        return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k,
           1], padding="SAME")
```

然后创建输入嵌入并使用随机值进行初始化：

```
with tf.name_scope('embedding'):
        embdded_chars =
tf.nn.embedding_lookup(params=tf.Variable(tf.random_normal([char_set_len,
char_embed_dim])), ids=tf_char_map, name='embedding')
        embedded_chars_expanded = tf.expand_dims(embdded_chars, -1)
```

在一端是扁平化层，接下来则要创建四层卷积，并逐步增加过滤器及步长。更大的步长意味着关注较长时间维度上的文本：

```
    prev_layer = embedded_chars_expanded
    with tf.name_scope('Character_CNN'):
        for idx, layer in enumerate([[3, 3, 1, 16], [3, 3, 16, 32], [3, 3,
32, 64], [3, 3, 64, 128]]):
            with tf.name_scope('Conv{}'.format(idx)):
                w = tf.Variable(tf.truncated_normal(layer, stddev=1e-1),
name='weights')
                b = tf.Variable(tf.truncated_normal([layer[-1]],
stddev=1e-1), name='bias')
                conv = conv2d(prev_layer, w, b)
                pool = maxpool2d(conv, k=2)
                prev_layer = pool

        prev_layer = tf.reshape(prev_layer, [-1, prev_layer.shape[1]._value
* prev_layer.shape[2].value * prev_layer.shape[3]._value])

    return prev_layer
```

接着创建一个函数，为每个问题对创建和连接先前描述的两个 CNN 层，并通过减小维度来创建三个全连接层以形成最终的激活：

```
def model(x1_pls, x2_pls, char_map, keep_prob):
    out_layer1 = character_CNN(x1_pls, char_map)
    out_layer2 = character_CNN(x2_pls, char_map)
    prev = tf.concat([out_layer1, out_layer2], 1)
```

该模型类似于暹罗网络[①]，能够同时训练两个编码器：

---

① 暹罗网络（siamese network）是一种特殊类型的神经网络，其中每类只从一个训练例子中学习，主要用于每个类中数据点较少的应用程序，也译为孪生网络。——译者注

```
with tf.name_scope('fc'):
        output_units = [1024, 512, 128, 2]
        for idx, unit in enumerate(output_units):
            if idx != 3:
                prev = tf.layers.dense(prev, units=unit,
activation=tf.nn.relu)
                prev = tf.nn.dropout(prev, keep_prob)
            else:
                prev = tf.layers.dense(prev, units=unit, activation=None)
                prev = tf.nn.dropout(prev, keep_prob)
    return prev
```

这样，通过让几个卷积层后接最大池化层，模型就建立好了。

### 6.2.3 训练

接下来，我们创建一个函数来训练数据。首先为问题对及其标签创建占位符。通过交叉熵 softmax 将先前模型的输出作为损失函数，使用 Adam 优化器实现模型权重的优化：

```
def train(train_x1, train_x2, train_y, val_x1, val_x2, val_y, max_sent_len,
char_map, epochs=2, batch_size=1024, num_classes=2):
    with tf.name_scope('Placeholders'):
        x1_pls = tf.placeholder(tf.int32, shape=[None, max_sent_len])
        x2_pls = tf.placeholder(tf.int32, shape=[None, max_sent_len])
        y_pls = tf.placeholder(tf.int64, [None])
        keep_prob = tf.placeholder(tf.float32) # dropout
```



接着，创建模型并进行 logit 计算，损失则通过 logit 和标签的独热编码计算得到。我们使用 Adam 优化器对损失进行优化，并设置学习率为 0.001。这样计算出了正确的预测和准确率，如下所示：

```
    predict = model(x1_pls, x2_pls, char_map, keep_prob)
    with tf.name_scope('loss'):
        mean_loss = tf.losses.softmax_cross_entropy(logits=predict,
onehot_labels=tf.one_hot(y_pls, num_classes))
    with tf.name_scope('optimizer'):
        optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
        train_step = optimizer.minimize(mean_loss)
    with tf.name_scope('accuracy'):
        correct_prediction = tf.equal(tf.argmax(predict, 1), y_pls)
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    saver = tf.train.Saver()
```

启动会话以初始化权重。对于每个周期，都将编码后的数据混洗并馈送到模型中。验证数据时也遵循相同的过程：

```
with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        train_indicies = np.arange(train_x1.shape[0])
        variables = [mean_loss, correct_prediction, train_step]
```

然后迭代周期。对于任意周期，都混洗训练数据以进行更为稳健的训练：

```
iter_cnt = 0
for e in range(epochs):
    np.random.shuffle(train_indicies)
        losses = []
        correct = 0
```

接下来，对批数据进行迭代并切片以用于训练和验证模型，如以下代码所示：

```
for i in range(int(math.ceil(train_x1.shape[0] / batch_size))):
                start_idx = (i * batch_size) % train_x1.shape[0]
                idx = train_indicies[start_idx:start_idx + batch_size]
```

随后，生成 feed 字典并馈送到会话中，如以下代码所示：

```
feed_dict = {x1_pls: train_x1[idx, :],
             x2_pls: train_x2[idx, :],
             y_pls: train_y[idx],
             keep_prob: 0.95}
actual_batch_size = train_y[idx].shape[0]

loss, corr, _ = sess.run(variables, feed_dict=feed_dict)
```

使用计算后的损失和正确的重复对，可以得到准确率：

```
                corr = np.array(corr).astype(np.float32)
                losses.append(loss * actual_batch_size)
                correct += np.sum(corr)
                if iter_cnt % 10 == 0:
                    print("Minibatch {0}: with training loss = {1:.3g} and
accuracy of {2:.2g}" \
                          .format(iter_cnt, loss, np.sum(corr) /
actual_batch_size))
                iter_cnt += 1
            total_correct = correct / train_x1.shape[0]
            total_loss = np.sum(losses) / train_x1.shape[0]
            print("Epoch {2}, Overall loss = {0:.5g} and accuracy of
{1:.3g}" \
                  .format(total_loss, total_correct, e + 1))
```

每过 5 轮迭代，都准备验证数据并计算验证准确率，如下所示：

```
            if (e + 1) % 5 == 0:
                val_losses = []
                val_correct = 0
                for i in range(int(math.ceil(val_x1.shape[0] /
batch_size))):
                    start_idx = (i * batch_size) % val_x1.shape[0]

                    feed_dict = {x1_pls: val_x1[start_idx:start_idx +
batch_size, :],
                                 x2_pls: val_x2[start_idx:start_idx +
```

```
batch_size, :],
                                 y_pls: val_y[start_idx:start_idx +
batch_size],
                                 keep_prob: 1}
                    print(y_pls)
                    actual_batch_size = val_y[start_idx:start_idx +
batch_size].shape[0]
                    loss, corr, _ = sess.run(variables,
feed_dict=feed_dict)
                    corr = np.array(corr).astype(np.float32)
                    val_losses.append(loss * actual_batch_size)
                    val_correct += np.sum(corr)
```

计算匹配的准确率：

```
                total_correct = val_correct / val_x1.shape[0]
                total_loss = np.sum(val_losses) / val_x1.shape[0]
                print("Validation Epoch {2}, Overall loss = {0:.5g} and
accuracy of {1:.3g}" \
                      .format(total_loss, total_correct, e + 1))
            if (e+1) % 10 == 0:
                save_path = saver.save(sess, './model_{}.ckpt'.format(e))
                print("Model saved in path:{}".format(save_path))
```

保存模型用以在推理时进行还原，这将在下一节进行阐述。

### 6.2.4　推理

下面创建一个函数以对测试数据进行推理。该模型在上一步中已被存储为检查点，在这里可用于推理。以下代码定义了输入数据的占位符，还定义了一个 saver 对象：

```
def inference(test_x1, max_sent_len, batch_size=1024):
    with tf.name_scope('Placeholders'):
        x_pls1 = tf.placeholder(tf.int32, shape=[None, max_sent_len])
        keep_prob = tf.placeholder(tf.float32) # dropout

    predict = model(x_pls1, keep_prob)
    saver = tf.train.Saver()
    ckpt_path = tf.train.latest_checkpoint('.')
```

新建一个会话并还原模型：

```
with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        saver.restore(sess, ckpt_path)
        print("Model restored.")
```

将模型载入会话，传入批数据并储存预测结果：

```
        prediction = []
        for i in range(int(math.ceil(test_x1.shape[0] / batch_size))):
            start_idx = (i * batch_size) % test_x1.shape[0]
```

```
            prediction += sess.run([tf.argmax(predict, 1)],
                                   feed_dict={x_pls1:
test_x[start_idx:start_idx + batch_size, :], keep_prob:1})[0].tolist()
        print(prediction)
```

调用所有的函数以预处理数据、训练模型并在测试集上完成推理：

```
train_x1, train_x2, train_y, val_x1, val_x2, val_y, test_x1, test_x2,
max_sent_len, char_map = pre_process()
train(train_x1, train_x2, train_y, val_x1, val_x1, val_y, max_sent_len,
char_map, 100, 1024)
inference(test_x1, test_x2, max_sent_len)
```

当训练开始之后，你就可以看到训练过程及其结果，如下所示：

```
Validation Epoch 25, Overall loss = 0.51399 and accuracy of 1
Epoch 26, Overall loss = 0.19037 and accuracy of 0.889
Epoch 27, Overall loss = 0.15886 and accuracy of 1
Epoch 28, Overall loss = 0.15363 and accuracy of 1
Epoch 29, Overall loss = 0.098042 and accuracy of 1
Epoch 30, Overall loss = 0.10002 and accuracy of 1
Tensor("Placeholders/Placeholder_2:0", shape=(?,), dtype=int64)
```

在 30 个周期后，模型在验证集上已可以提供大约 100%的准确率。我们看到了如何使用 Quora 问题对为示例训练一个发现重复的模型。

## 6.3 小结

在本章中，我们通过训练模型来搜索重复对了解了如何使用基于字符的 CNN 模型。字符型 CNN 使我们可以灵活地训练具有未知字符的模型。相较于单词级嵌入，它更为通用。类似的网络可以用于搜索、匹配和去重。

在下一章，我们将学习如何使用 LSTM 训练模型进行命名实体识别。

第 7 章

# 使用字符级 LSTM 进行命名实体识别

执行重复性任务时，人类由于肌肉记忆和注意力缺失，很容易犯错误。人的大脑倾向于自动操作而不考虑动作和反应，此时注意力就会不集中，这通常称为**大脑疲劳**。因此，要在不丢失任何信息或发生任何错误的情况下回答问题，就迫切需要改进传统的用户接口，从根本上改变与机器交互的方式。这些用户接口对客户服务、搜索接口和人机交互的大量应用都有影响，因此也是一个非常重要的研究领域。

为了开发此类接口，基本任务之一是理解并解释用户输入的句子。此类接口应该能够识别句子中的单词及其传达给用户的含义。这样的过程称为**命名实体识别**（named entity recognition，NER），目标是在文本中查找（并且分类）命名实体。命名实体识别属于更为广泛的信息检索领域，通常以**实体标识**、**实体分块**和**实体提取**之类的名称为人熟知。

在 NER 中，实体是预定义的类别，例如人员名称、组织名称、位置名称、时间，等等。NER 允许计算机程序将句子“I will meet you at Burj Khalifa for a cup of coffee at 7:30 PM tomorrow”理解为“I will meet you at $(\text{Burj Khalifa})_{\text{Location}}$ for a cup of $(\text{coffee})_{\text{Food}}$ at $(\text{7:30 PM})_{\text{Time}}$ $(\text{tomorrow})_{\text{Date}}$”。在本例中，该算法检测并分类出了一个双词元的位置、一个单词元的食物以及一个时间表达式和一个日期。

NER 通常被认为属于序列标记问题，使用**隐马尔可夫模型**（HMM）、**决策树**、**最大熵**（ME）模型、**支持向量机**（SVM）和**条件随机场**（CRF）等方法。但在最近的文献中，深度学习已被广泛用于识别命名实体。深度学习借助大量数据来构建算法，展现出了胜过传统方法且同时也具有很好的学习泛化能力。

必须指出，基于英语的先进 NER 系统已经产生了接近人类的表现。例如最佳系统的 F 度量得分为 93.39%，而人类标注者的得分约为 97%。

## 7.1 使用深度学习实现 NER

深度学习提供了一个利用大量数据为 NER 提取最佳特征的好机会。NER 的深度学习方法通

常使用 RNN，因为该问题属于序列标记任务。RNN 具有处理可变长度输入的能力。它的变体称为 LSTM，具有长期记忆，对于理解给定句子中单词的重要依存关系很有用。双向 LSTM 则是 LSTM 的变体，不仅具有理解长期依赖性的能力，还具有从句子两端理解句中单词间关系的能力。

本章将使用 LSTM 进行深度学习来构建 NER 系统。但在尝试之前，我们先来看看深度学习模型中将要使用的数据（以及对应数据的格式）。

### 7.1.1　数据

最常用来测试和基准化 NER 的数据是 CoNLL2003 数据集，这是一个与语言无关的 NER 共享任务。该数据集里有用于训练、开发和测试的文件以及一个包含未标注数据的大型文件。开发文件用于调整学习方法的参数，而训练数据则用于训练模型、使用调整后的参数并在测试数据集上进行测试。

CoNLL 数据在开发和测试之间做了划分，以避免对测试数据进行系统调整，其中的英文数据取自路透社语料库从 1996 年 8 月至 1997 年 8 月的新闻报道。CoNLL 数据集的示例句子及其附带的实体注释如下所示：

```
Only RB B-NP O
France NNP I-NP B-LOC
and CC I-NP O
Britain NNP I-NP B-LOC
backed VBD B-VP O
Fischler NNP B-NP B-PER
' s POS B-NP O
proposal NN I-NP O
. . O O
```

CoNLL 数据的每一行包含四个字段：单词、单词的**词性**（POS）标签、单词的块标签以及单词的命名实体标签，其中标签 `O` 被赋给了命名实体之外的单词。

为了处理存在二元词元的实体（例如 New York），一种**标记方案**被用来区分不同的实体情况。当`<entity>`类型的两个实体彼此相邻时，第二个实体的第一个单词被标记为 `B- <entity>`，以表明它启动了另一个实体。CoNLL2003 任务提供的实体分别是 `LOC`、`PER`、`ORG` 和 `MISC`，分别是位置、人员、组织和其他实体。

另一个通常用于构建 NER 系统的数据集是 Groningen Meaning Bank（GMB）。它具有许多对构建 NER 系统任务有用的标注，但是本章仍使用 CoNLL2003 数据来进行实验和评估。

用于现成 NER 系统的流行开源框架有斯坦福大学的 NLTK 和 Explosion AI 的 spaCy。尽管 这两个框架在几个任务上都有优秀的表现，但我们仍对开发灵活、先进的深度学习模型来实现 NER 系统感兴趣。

## 7.1.2 模型

如前所述，我们可以将命名实体识别问题视为序列问题。大多数 NLP 系统遵循的通用深度学习方法是使用 RNN。但在决定 RNN 模型的架构之前，我们需要考虑如何提供模型输入和处理模型输出。

由于我们的输入数据为单词形式，每个单词 $w \in \mathbb{R}^n$ 都需要一个密集向量表示（即**词嵌入**）。常用的预训练词嵌入方法包括 Word2vec、GloVe 和 fastText。这样的预训练词向量为每个单词提供了非常好的语义表示，从而无须为每个任务去单独训练它们。因为许多实体没有预训练的词向量，所以我们可以提取字符级向量表示形式，以考虑出现连字符或以大写字母开头的单词等情况。这种做法为模型提供了有关当前上下文所讨论实体的有价值信息。

当决定使用 RNN 的变体（例如 LSTM）来获取上下文中输入的语义表示时，我们将为每个单词获取一个向量表示，以便对实体进行预测。

**词嵌入**

正如在第 4 章中讨论的那样，我们想构建一个密集的向量来捕获单词上下文的语义。但在此任务中，将把词嵌入构建为在单词级别提取的预训练嵌入和从字符级别提取的训练嵌入的串联。因此，词嵌入 $w \in \mathbb{R}^d$ 由预训练的单词级向量 $w_{pretrain} \in \mathbb{R}^{d_1}$ 和一个预训练的字符级向量 $w_{char} \in \mathbb{R}^{d_2}$ 组成。

尽管可以将字符级向量编码为独热编码或使用其他手工特征，但是这样得到的特征可能不容易扩展到其他数据集和语言。一种可靠的字符级编码方法是直接从数据中学习。本章将使用双向 LSTM 来学习该嵌入，如图 7-1 所示。

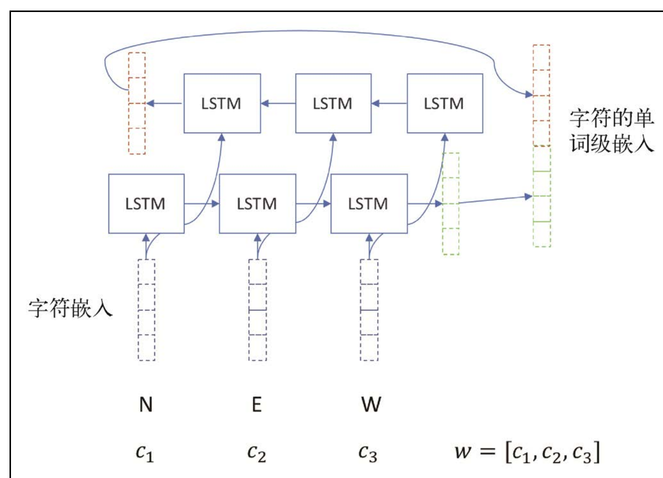



图 7-1

因此，单词 $w=[c_1,c_2,\cdots,c_k]$中的每个字符 $c_i$都有关联向量 $c_i\in\mathbb{R}^{d_2}$。必须注意，我们不对数据执行任何预处理（如删除标点符号或将单词更改为小写），因为此类字符会影响给定位置单词所传达的含义。例如句子“They had to get an Apple product to complete their tech eco-system”中的 Apple 一词指的是组织（ORG），而句子“They had to get an apple a day to keep the doctor away”中的 apple 指的则是水果。当考虑单词的大写用法时，可以轻松识别单词的区别。字符级嵌入向量试图学习结构和单词以得出最终的使用形式。换句话说，字符级向量学习的是对应单词的词法。

最后，我们将字符嵌入 $w_{char}\in\mathbb{R}^{d_2}$ 词嵌入 $w_{pretrain}\in\mathbb{R}^{d_1}$ 串联起来，以获得单词的表示形式 $w=[w_{pretrain},w_{char}]\in\mathbb{R}^{d}$，其中 $d=d_1+d_2$。现在我们已经准备好了输入，下面继续遍历代码以构建输入和模型。

### 7.1.3　代码详解

本节将详细解析 TensorFlow 的代码以构建输入。以下各小节将遍历代码的不同部分。

#### 1. 输入

我们正在考虑动态输入，因此将使用 TensorFlow 的**占位符**。因为关于批尺寸和每批单词数，我们都有明确的数据形状，所以需要先填充数据集中的句子，使它们具有相同的长度。因此，我们将定义两个占位符，如下所示：

```
# 定义单词索引形状的输入占位符 = (批尺寸, 填充句子的长度)
word_indices = tf.placeholder(tf.int32, shape=[None, None])

# 序列长度形状的占位符 = (批尺寸)
sequence_lengths = tf.placeholder(tf.int32, shape=[None])
```

序列长度占位符允许计算图利用输入数据的实际长度，因为我们对初始输入进行了填充以使其具有相同的 `max_length`。

#### 2. 词嵌入

现在我们已经定义了输入占位符，接下来将定义一个 TensorFlow `Variable` 来保存数据中词汇的预训练嵌入。在这种情况下，我们将采用索引数组，将与 `i` 处的单词索引相对应的嵌入作为 `pre_trained_embedding[i]`供获取。因为我们想从嵌入矩阵中查找，所以将预训练的嵌入数组加载到 TensorFlow 变量中，其代码定义如下：

```
# 使用预训练嵌入矩阵定义查找表
lookup_word_mat = tf.Variable(embedding_matrix, dtype=tf.float,
trainable=False)

# 使用TensorFlow内置函数定义嵌入查找表
pre_trained_embedding = tf.nn.embedding_lookup(lookup_word_mat,
word_indices)
```

在以上代码块中，我们使用 `trainable = False` 来定义 TensorFlow 变量，因为不希望该算法进一步训练嵌入。该代码块定义了单词级别的向量表示。

我们将以类似的方法使用两个占位符构建字符级向量表示，如下所示：

```
# 定义字符索引形状的输入占位符 = (批尺寸, 句子最大长度, 单词最大长度)
char_indices = tf.placeholder(tf.int32, shape=[None, None, None])

# 单词长度形状的占位符 = (批尺寸, 句子最大长度)
word_lengths = tf.placeholder(tf.int32, shape=[None, None])
```

可以看到，我们选择根据每次迭代过程中处理批次里的可用数据来动态设置句子和单词的长度。在定义了如何使用字符来构建词嵌入后，我们将研究如何训练嵌入。

与之前的情况不同，我们在字符级别没有任何预训练的嵌入可供在嵌入矩阵中查找。因此，我们将使用随机向量初始化字符嵌入，如下所示：

```
# 使用默认初始器定义变量查找表
lookup_char_mat = tf.get_variable(name="character_embeddings",
dtype=tf.float32, shape=[num_characters, dim_character])

# 使用 TensorFlow 内置函数定义嵌入查找表
character_embedding = tf.nn.embedding_lookup(lookup_char_mat, char_indices)
```

如图 7-1 所示，我们将定义一个双向 LSTM 以从可用输入的各批数据里获取字符：

```
# 为前向 RNN 定义 LSTM 以接受字符
bi_dir_cell_fw = tf.contrib.rnn.LSTMCell(char_hidden_dim,
state_is_tuple=True)

# 为反向 RNN 定义 LSTM 以接受字符
bi_dir_cell_bw = tf.contrib.rnn.LSTMCell(char_hidden_dim,
state_is_tuple=True)

# 定义双向 LSTM 以从前向 RNN 和反向 RNN 中获取输入及序列长度
_, ((_, out_fw), (_, out_bw)) =
tf.nn.bidirectional_dynamic_rnn(bi_dir_cell_fw, bi_dir_cell_bw,
character_embedding, sequence_length=word_lengths, dtype=tf.float32)
```

因为我们只需要输出向量，所以不存储 `bidirectional_dynamic_rnn` 函数提供的其余返回语句。为了导出输出，我们将连接所产生的前向和后向输出，从而得到一个原始字符隐藏维度大小两倍的输出维度。因此我们将通过重新调整所连接的输出来获得字符和单词的表示形式：

```
# 连接前向 RNN 和反向 RNN 的两个输出向量，从而得到形状为(批尺寸×句子，2×char_hidden_dim)的向量
output_fw_bw = tf.concat([out_fw, out_bw], axis = -1)

# 通过改变先前步骤中输出向量的形状得到字符嵌入，最终向量形状 = (批尺寸, 句子长度, 2 ×
char_hidden_dim)
char_vector = tf.reshape(output_fw_bw, shape=[-1,
tf.shape(character_embedding)[1], 2*char_hidden_dim])
```

```
# 通过将预训练词嵌入及由双向 LSTM 得到的字符级嵌入级联得到最终词嵌入
word_embedding = tf.concat([pre_trained_embedding, char_vector], axis = -1)
```

### 7.1.4　不同预训练词嵌入的影响

有许多预训练的词嵌入可以为我们所用。实际上，它们包括单词和由不同研究团队制作的相应 $n$ 维词向量。著名的预训练词向量包括 GloVe、Word2vec 和 fastText。尽管前面这些词嵌入对于构建本章所讨论的 NER 系统都是有用的，但此处还是使用预训练的 Word2vec 词向量。这些预训练词嵌入模型的读取和处理方法有所区别，如图 7-2 所示。

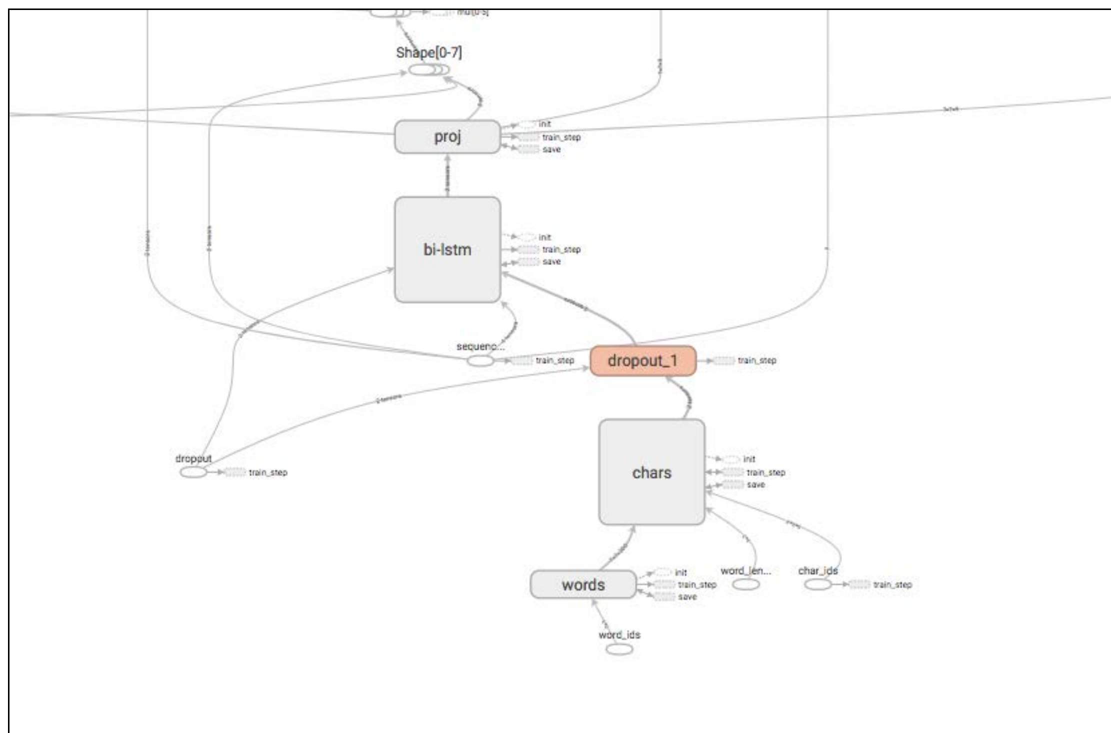



图 7-2　TensorBoard 图

该 TensorBoard 图显示了单词和字符是如何被用作双向 LSTM 的输入的。

#### 1. 神经网络架构

现在我们已经从字符–词嵌入连接中构建了词向量，下面将在词嵌入序列上运行双向 LSTM，并使用双向 LSTM 的隐藏连接状态（向前和向后）获得嵌入语义表示，如图 7-3 所示。

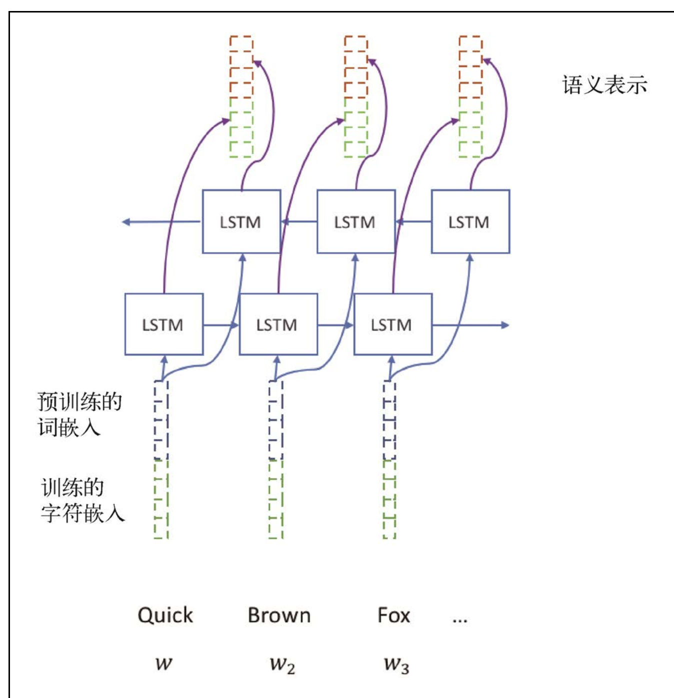



图　7-3



使用 TensorFlow 来实现这一点很简单，并且非常类似于我们之前实现的字符级嵌入学习 LSTM 的方式。但与之前情况不同的是，我们对每个时间步骤的隐藏状态感兴趣：

```
bi_dir_cell_fw = tf.contrib.rnn.LSTMCell(hidden_state_size)

bi_dir_cell_cell_bw = tf.contrib.rnn.LSTMCell(hidden_state_size)

(out_fw, out_bw), _ = tf.nn.bidirectional_dynamic_rnn(bi_dir_cell_fw,
bi_dir_cell_bw, word_embedding, sequence_length=sequence_lengths,
dtype=tf.float32)

semantic_representation = tf.concat([out_fw, out_bw], axis=-1)
```

因此，语义表示 $h$ 使用预训练的词向量、字符和单词上下文来捕获每个可用单词 $w$ 的含义。有了这样的向量，我们可以使用密集神经网络来获取预测向量，且向量中的每个元素对应于我们想要作为实体去预测的标签，如图 7-4 所示。

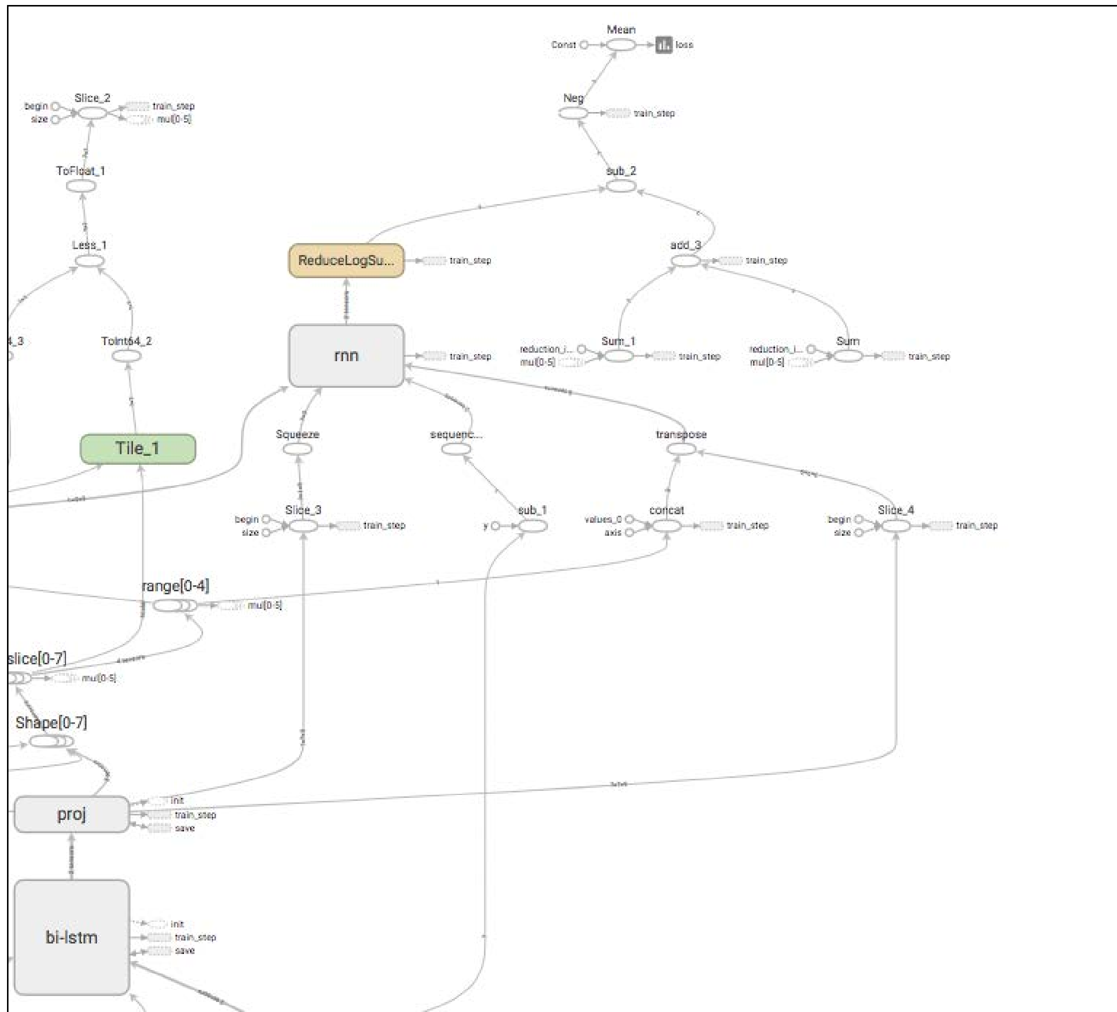



图 7-4　语义图表示

图 7-5 显示了双向 LSTM 的输出是如何馈入的。

图 7-5 双向 LSTM

### 2. 解码预测

假设我们想实现五分类任务，此时将计算一个五维向量 $s \in \mathbb{R}^5 = \mathbb{W}.h + b$。可以将该向量解释为属于每个类的概率。这意味着在给定单词 $w$ 的情况下，$s$ 的第 $i$ 个分量提供了第 $i$ 类的概率或分数。



密集向量可以在 TensorFlow 中计算，如以下代码所示：

```
# 使用默认初始器初始化权重矩阵
W = tf.get_variable("W", shape=[2 * hidden_state_size, ntags],
dtype=tf.float32)

# 使用零初始器初始化偏差向量
b = tf.get_variable("b", shape=[ntags], dtype=tf.float32,
initializer=tf.zeros_initializer())

# 获取时间步数
num_time_steps = tf.shape(semantic_representation)[1]

# 使用扁平（用 reshape 得到）向量计算预测得分
prediction = tf.matmul(semantic_representatio_flatten, W) + b
```

假设我们可以计算出一个提供给定单词属于第 $i$ 类的得分的密集向量，那就可以使用两种方法进行预测。

第一种方法显而易见，是使用 softmax 激活将分数标准化为向量 $p \in \mathbb{R}^5$。此方法使用非线性激活，其中向量的元素计算公式如下：

$$p[i] = \frac{e^{s[i]}}{\sum_{j=1}^{5} e^{s[j]}}$$

第二种方法则更为聪明一些。它使用类似于第一步的方式获得相邻标签，然后用其来标记单词。例如，如果我们考虑使用 New Delhi 一词，NER 系统可能会基于 Delhi 是地点的事实，将其附近的 New 一词归为某个地点的开头。此方法也称为线性链**条件随机场**（**CRF**），它定义了一个全局分数，而该分数考虑了以特定标签开头或结尾的成本。

### 3. 训练步骤

在前面的两小节中，我们讨论了输入、网络架构以及如何对隐藏状态解码以预测输出向量。现在，我们将定义用于训练开发模型的对象函数。

在本例中，采用交叉熵损失是十分有效的。我们将使用第二种方法，使用 CRF 解码输出状态。不过，要实现这样一个复杂的概念，我们将依靠 TensorFlow 提供的易于上手的函数，如以下代码所示：

```
# 定义标签占位符，形状 = (批尺寸, 句子)
labels = tf.placeholder(tf.int32, shape=[None, None], name="labels")

log_likelihood, transition_params = tf.contrib.crf.crf_log_likelihood(
scores, labels, sequence_lengths)
```

随着 TensorFlow 的出现，CRF 的实现变得非常简单。我们终于可以使用刚刚计算的对数似然值来计算损失了，如下所示：

```
# 对对数似然值取反以获得距离度量
loss = tf.reduce_mean(-log_likelihood)
```

为获得最终分数，一个经典方法是通过实现 `softmax` 方法来计算损失值。然而，我们需要注意填充值（padding）：

```
loss_values =
 tf.nn.sparse_softmax_cross_entropy_with_logits(logits=scores,
 labels=labels)

loss = tf.reduce_mean(loss_values)
```

最后，使用我们刚刚定义的损失函数来完成训练。在本例中，我们选择使用 `AdamOptimizer`（尽管其他优化器也是可以的）来最小化损失值：

```
opt = tf.train.AdamOptimizer(learning_rate)

training_op = opt.minimize(loss)
```

使用 CRF 方法，可以获得大约 88%的 F1 值，而准确率约为 92%，但这种方法依赖于预训练词嵌入。因此我们将训练另一个网络，其中所有词嵌入都是从头训练的。图 7-6 显示了这两种方法的训练损失。

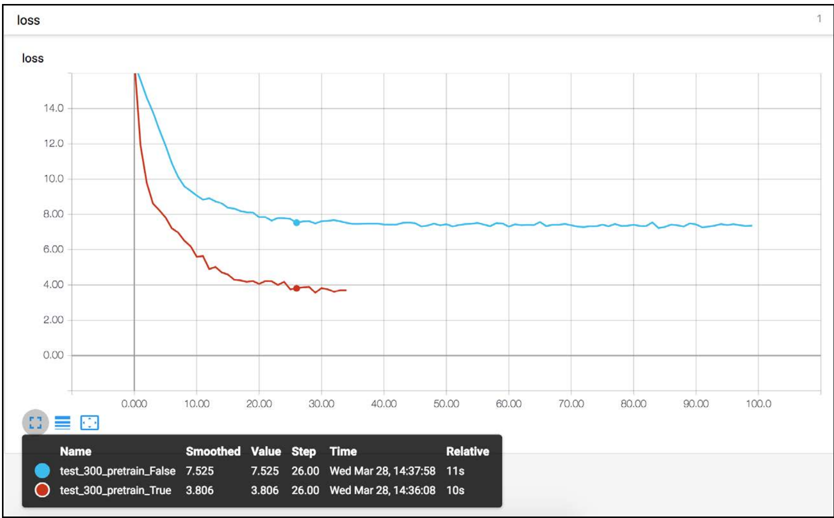



图 7-6　训练损失图

在这次训练中，我们让模型训练了 100 个周期，并监控其训练损失。如果在一定周期内验证准确率没有提高，我们将检视验证集并退出训练。当我们使用预训练的词嵌入时，损失大大下降。因为性能不再提升，训练在第 35 个周期就停止了。在不使用预训练的词嵌入的情况下进行训练，最终准确率约为 69%。

先前训练步骤所使用的词嵌入大小维度为 300，但是较大的维度提出了较大的存储要求，而当我们进行推理时就需要较长的加载时间。因此，再以维度 100 重复前面的实验，如图 7-7 所示。

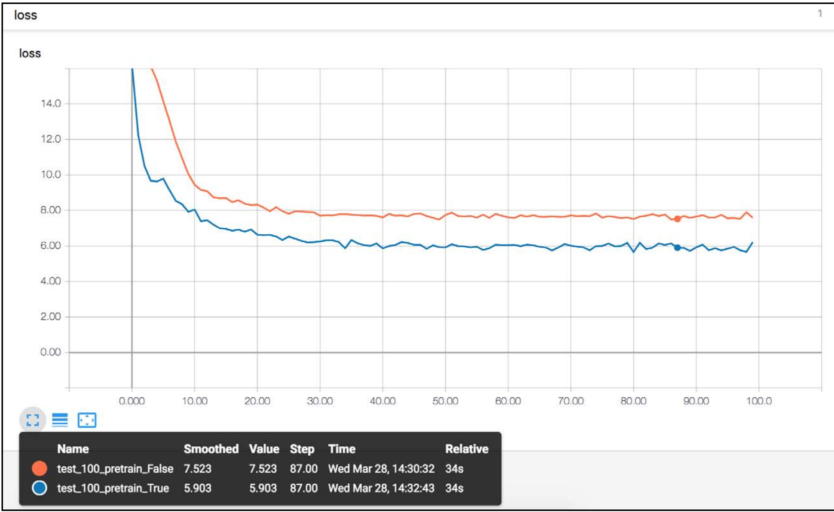



图 7-7　训练损失图

在减少词嵌入维度的情况下，词的表示任务变得具有挑战性。尽管它减少了训练模型的大小（以及初始的预训练词嵌入文件），但会导致数据丢失。

使用预训练的词嵌入向量（维度 100）进行训练可显著降低训练损失，并且最终测试准确率约为 74%。与维度为 300 时的预训练词嵌入相比下降了 18%。类似地，对未经预训练的词嵌入向量进行训练会产生更低的准确率，约为 64%，如图 7-8 所示。

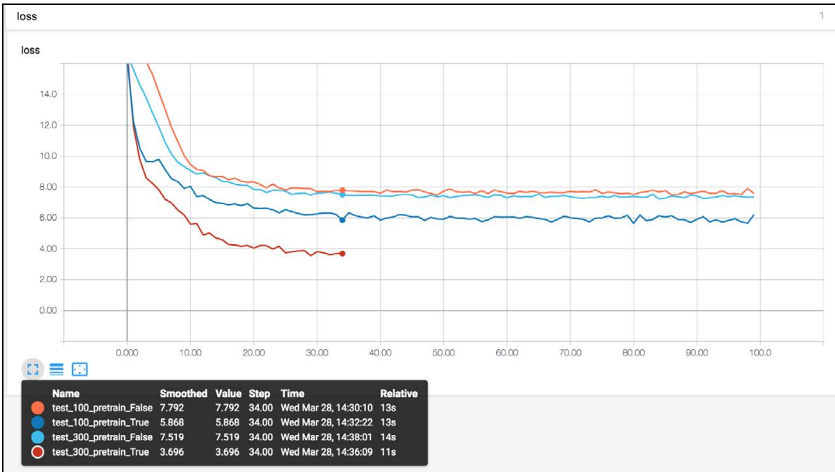



图 7-8　训练损失图

当我们比较维度分别为 300 和 100 的模型的性能时，可以更好地了解两种方法对模型性能的影响。相较于 100 的维度，维度为 300 时明显能够提取更多的信息。同样，使用预训练词嵌入可大大减少训练时间，并带来更好的模型性能，如图 7-9 所示。

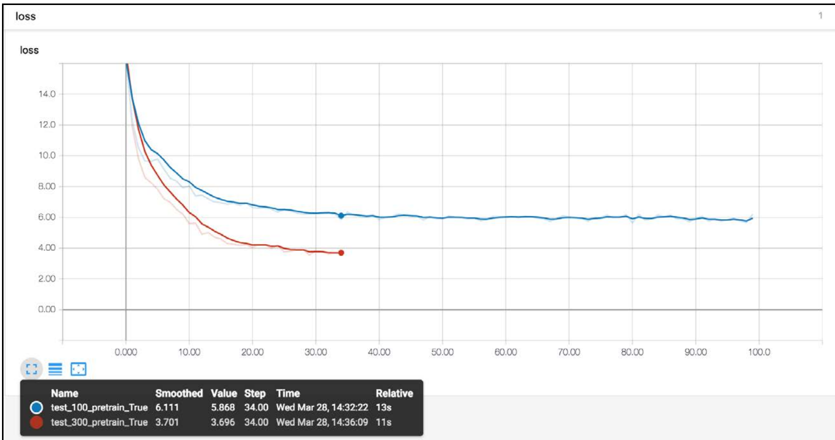



图 7-9　训练损失图

当我们比较最佳性能模型在不同维度上的性能时，可以很明显地看到，词嵌入维度为 300 时所训练的模型性能要好得多。在第 34 个周期时，该模型不仅具有较低的损失值，且其训练速度也快得多，如图 7-9 所示。

我们可以使用经典的 softmax 激活（而非 CRF）执行另一项评估以进行最终评分。为了客观评估使用 CRF 的优势，我们将使用具有 300 个维度的预训练模型，这是目前效果最好的模型，如图 7-10 所示。

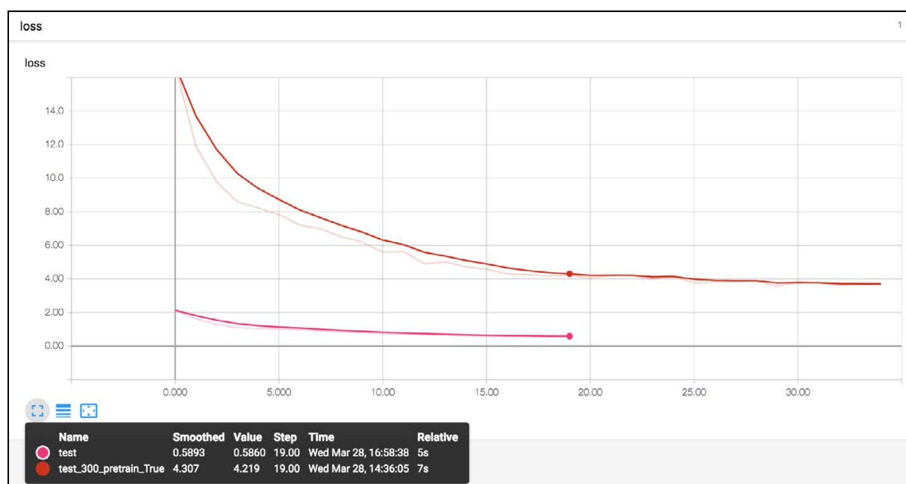



图 7-10 训练损失图

图 7-10 比较了两种方法的训练过程。基于 softmax 的方法可实现约 74%的准确率，并且仅训练 20 个周期即可完成。但就性能而言，观察训练后的准确率得分可以看出，基于 softmax 的方法不如基于 CRF 的模型。图中反映的较低损失值是由 CRF 和 softmax 损失指标的差异造成的。

### 7.1.5 改进空间

尽管本章讨论的框架对于现有数据非常有效，但是当数据集较小或数据不平衡时，情况可能并非如此。

## 7.2 小结

本章实现了一个字符级、基于 LSTM 的神经网络以开发一种检测命名实体的算法，展示了如何通过 TensorFlow 轻松实现复杂的方法，并研究了改善模型性能的方法。

在下一章，我们将讨论如何开发用于文本生成的深度神经网络。

第 8 章

# 使用 GRU 进行文本生成和文本摘要

本章将介绍使用深度学习技术来生成文本和摘要的方法。文本生成是通过使用输入源文本，根据上下文和范围来自动生成文本的过程。一些涉及文本生成的应用包括自动生成天气报告，生成医疗报告，以及将给定输入文本的表示形式翻译成多种语言。文本摘要是一种与之相关技的术，涉及从源文本生成摘要，其示例任务包括生成新闻、产品评论和业务报告摘要。

本章着重为你提供使用 RNN 进行文本生成和文本摘要的方法，涵盖以下主题：

- 使用 RNN 进行文本生成
- 使用 RNN 进行文本摘要
- 总结先进的文本生成和文本摘要技术

## 8.1 使用 RNN 进行文本生成

在前面的章节中，我们使用了 LSTM 和 GRU 进行文本分类。除了预测任务外，RNN 还可用于创建生成模型。它可以从输入文本中学习长期依赖关系，以此生成全新的序列。该生成模型可以是基于字符或基于单词的。在下一节，我们将学习一个基于单词的简单文本生成模型。

### 使用GRU生成Linux内核代码

现在，我们将看一个简单有趣的示例：使用 GRU（RNN 网络的变种）生成 Linux 内核代码。该示例的完整 Jupyter Notebook 可在本书代码库中的文件夹 Chapter08 下找到。对于训练数据，我们将首先从 Linux 源代码中提取内核代码。你可以从内核档案（The Linux Kernel Archives）里下载其最新版本（或更早版本）。

解压缩 tar 文件，仅使用源代码中 kernel 目录下的核心内核。在内核树根目录执行以下命令，以从所有*.c 文件中提取代码：

```
cd kernel
find . -name "*.c" -exec cat &gt;&gt; /tmp/kernel.txt {} \;
```

这将连接核心 kernel 目录下的所有*.c 文件并将其写入/tmp 下的 kernel.txt 文件中。当然，你也可以使用/tmp 目录以外的任何目录。首先，从原始内核代码文件中准备训练数据：

```
with codecs.open('/tmp/kernel.txt', 'r', encoding='utf-8', errors='ignore')
as kernel_file:
    raw_text = kernel_file.read()
kernel_words = re.split('(\-\&gt;)|([\-
\&gt;+\=\&lt;\/\&\|\(\)\:\*])',raw_text)
kernel_words = [w for w in kernel_words if w is not None]
kernel_words = kernel_words[0:300000]
kernel_words = set(kernel_words)
kword_to_int = dict((word, i) for i, word in enumerate(kernel_words))
int_to_kword = dict((i, word) for i, word in enumerate(kernel_words))
vocab_size = len(kword_to_int)
kword_to_int['&lt;UNK&gt;'] = vocab_size
int_to_kword[vocab_size] = '&lt;UNK&gt;'
vocab_size += 1
X_train = [kword_to_int[word] for word in kernel_words]
y_train = X_train[1:]
y_train.append(kword_to_int['&lt;UNK&gt;'])
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)
X_train = np.expand_dims(X_train,axis=1)
y_train = np.expand_dims(y_train,axis=1)
print(X_train.shape, y_train.shape)
```

在代码中，正则表达式 re.split('(\-\>)|([\-\>+\=\<\/\&\|\(\)\:*])', raw_text)拆分了包含某些 C 语言运算符的语句，例如指针、算术和逻辑运算符。尽管在使用字符级文本生成器时这并非必需，但由于要生成单词级别的文本，我们在这里还是要使用它。我们还将创建一个字典，分别在变量 kword_to_int 和 int_to_kword 中将单词映射到整数 ID（反之亦然）。NumPy 变量 X_train 和 y_train 分别保存训练数据和标签。在 X_train 中，我们有一个从 kword_to_int 提取的单词 ID 列表，而 y_train 则包含 X_train 中每个单词的后续单词。使用 numpy.expand_dims 函数将这些 NumPy 数组重构为单词 ID 的一维向量。

与之前章节中一样，我们将使用 TensorFlow estimator API 进行模型训练和验证。

(1) 先来看看 model 函数的代码：

```
def rnn_model_fn(features, labels, mode):
    embedding = tf.Variable(tf.truncated_normal([v_size,
EMBED_DIMENSION],
stddev=1.0/np.sqrt(EMBED_DIMENSION)),
                                name="word_embeddings")
    word_emb = tf.nn.embedding_lookup(embedding, features['word'])
    rnn_cell = tf.nn.rnn_cell.GRUCell(HIDDEN_SIZE)
    outputs, _ = tf.nn.dynamic_rnn(rnn_cell, word_emb,
dtype=tf.float32)
```

```
    outputs = tf.reshape(outputs, [-1, HIDDEN_SIZE])
    flayer_op = tf.layers.dense(outputs, v_size, name="linear")
    return estimator_spec_for_generation(flayer_op, labels, mode)
```

我们使用了一个简单网络，在输入嵌入层后接上一个 GRUcell 和一个密集层。



请注意，EMBED_DIMENSION 和 HIDDEN_SIZE 分别被定义为 50 和 256。你也可以使用不同的值来尝试生成的输出。

(2) 然后将密集层的输出馈送到评估器特定函数所定义的优化器里，下面将进行探讨：

```
def estimator_spec_for_generation(flayer_op, lbls, md):
    preds_cls = tf.argmax(flayer_op, 1)
    if md == tf.estimator.ModeKeys.PREDICT:
        prev_op = tf.reshape(flayer_op, [-1, 1, v_size])[:, -1, :]
        preds_op = tf.nn.softmax(prev_op)
        return tf.estimator.EstimatorSpec(
        mode=md,
        predictions={
            'preds_probs': preds_op
        })
    trng_loss = tf.losses.sparse_softmax_cross_entropy(labels=lbls,
logits=flayer_op)
    if md == tf.estimator.ModeKeys.TRAIN:
        optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
        trng_op = optimizer.minimize(trng_loss,
global_step=tf.train.get_global_step())
        return tf.estimator.EstimatorSpec(md, loss=trng_loss,
train_op=trng_op)
    ev_met_ops = {'accy': tf.metrics.accuracy(labels=lbls,
predictions=preds_cls)}
    return tf.estimator.EstimatorSpec(md, loss=trng_loss,
train_op=trng_op)
```

在训练期间，我们使用 AdamOptimizer 以最大程度减少 sparse_softmax_cross_entropy 的损失，并使用标签指定序列中的下一个单词。在预测期间，将 softmax 输出作为下一个单词的概率。此 softmax 输出表示由词汇表里所有长度为 v_size 的单词所组成的列表中下一个单词的概率分布。

(3) 接着创建用于训练的 estimator，并设置训练配置：

```
run_config = tf.contrib.learn.RunConfig()
run_config = run_config.replace(model_dir='/tmp/models/',
             save_summary_steps=10,log_step_count_steps=10)
generator =
  tf.estimator.Estimator(model_fn=rnn_model_fn,config=run_config)
```

我们配置了步骤摘要日志，因此它每隔 10 步训练就进行一次计数。然后，我们使用 model 函数和配置创建了 estimator。

(4) 现在训练模型：

```
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'word': X_train},
    y=y_train,
    batch_size=1024,
    num_epochs=None,
    shuffle=True)
generator.train(input_fn=train_input_fn, steps=300)
```

(5) 创建训练输入函数 `train_input_fn`，使用输入训练数据 `X_train` 和 `y_train`。我们将批尺寸设置为 `1024`，并训练模型达 300 步。执行该模型后，将在输出中达到以下损失：

```
INFO:tensorflow:global_step/sec: 0.598131
INFO:tensorflow:Saving checkpoints for 300 into
/tmp/models/model.ckpt.
INFO:tensorflow:Loss for final step: 0.0061470587.
```

(6) 最后，使用经过训练的模型来生成文本，一次生成一个单词。为此，我们将使用 `generator` 来预测给定初始单词的下一个单词。预测的单词将被用作输入以预测后续单词，依此类推。我们将所有这些预测单词连接为最终生成的文本：

```
maxlen = 40
next_x = X_train[0:60]
text = "".join([int_to_kword[word] for word in next_x.flatten()])
for i in range(maxlen):
    test_input_fn = tf.estimator.inputs.numpy_input_fn(
      x={'word': next_x},
      num_epochs=1,
      shuffle=False)
    predictions = generator.predict(input_fn=test_input_fn)
    predictions = list(predictions)
    word = int_to_kword[np.argmax(predictions[-1]['preds_probs'])]
    text = text + word
    next_x = np.concatenate((next_x,[[kword_to_int[word]]]))
    next_x = next_x[1:]
```

我们应该选择随机单词序列作为初始文本。这里选择了前 60 个单词，你当然也可以从原始内核代码中选择其他任何连续的单词列表。我们将其存储在 `next_x` 变量中以跟踪序列中的下一个单词。在数据准备期间所创建的 `int_to_kword` 词典将预测的 ID 转换为单词并附加到 `text` 输出变量之后。请注意，我们在代码中将循环数 `maxlen` 设置为 40。你可以对其进行调整，以尝试增加或减少生成文本中的单词数。

现在可以看看最终生成的输出：

```
static int blk_trace_remove_queue Initialize POSIX timer handling for a
thread group.
  PAGE_SHIFT;
    if !rb_threads[cpu]s, const struct pci_dev ;
  check_mm the filter_hash does not exist or is empty,
```

```
    return NULL;

  memsetkexec_image;
struct kimage ;
}

power_attr_rostruct hist_field module_add_modinfo_attrs Fake ip  data;
  struct page
{
  return arg ? GFP_ATOMIC  PM_SUSPEND_ON
    goto fail_free_buffers;

  ret  sec;
  }

static struct ftrace_ops trace_ops __initdata  into them directly.
   !is_sampling_eventholders_dir, mod MIN_NICE can be offsets in the trace
data.
  to the buffer after this will fail and return NULL.
  ring_buffer_record_enable_cpu  {
    area[pos] ;

#ifdef CONFIG_SUSPEND
  if  hist_field_u16;
    break;
  case 1 representing a file path of format and ;

#endif ;
    goto out;
  }

  ftrace_graph_return  Pipe buffer operations for a buffer.  val;
  arch_spin_unlock If we fail, we do not register this tracer.
  return ret;
}
```

尽管输出与内核代码非常相似，但其实际意义并不大。请注意，每次运行的输出可能会有所不同，并且也会与本书代码库的 Notebook 中有所区别。

看起来该模型已经学习了一些经典的 Linux 内核习惯用法，例如对 `goto` 的使用等。使用更深层的网络和双向 LSTM 等可以通过字符级模型生成更实际的内核代码。你可以尝试使用这种方法获得更好的结果。

现在，我们将在 TensorBoard 中查看损失函数和模型图。首先看看模型图，如图 8-1 所示。

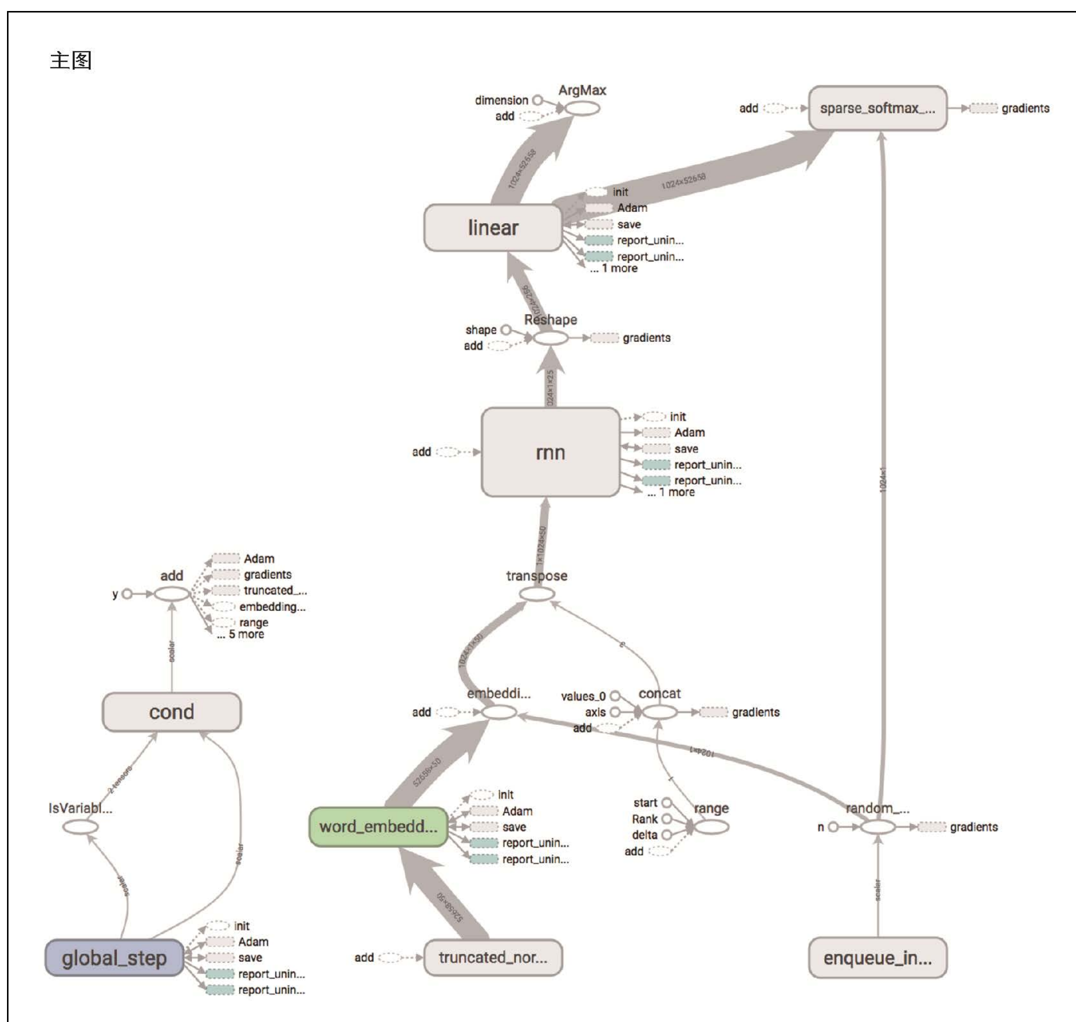



图 8-1 TensorBoard 中的损失函数及模型

该图显示了词嵌入的输入，其后是 RNN 单元和密集层。稀疏的 softmax 层提供输入词汇表中单词的最终输出概率。梯度将被输入到损失函数中，`AdamOptimizer` 则尝试最小化该损失函数。现在我们将研究损失函数在训练步骤中的变化，如图 8-2 所示。

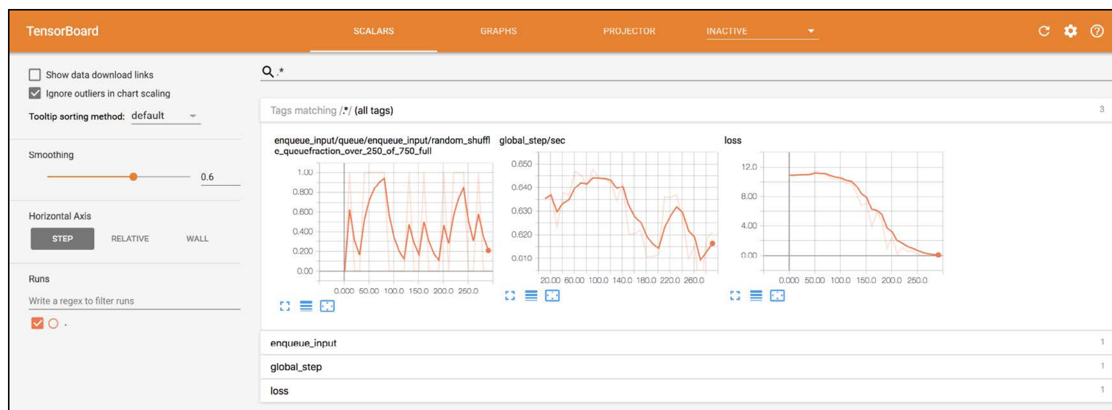



图 8-2　训练中损失函数变化曲线

可以看到，损失从 11 开始，慢慢减少到 0.1。

尽管本例本身并没有实际用处，但其目的是展示这种可用于学习输入文本的生成模型的基本架构和本质。因此，创建此类序列模型可以帮助我们了解给定领域的文本结构。我们在示例中使用的领域是操作系统代码，而你可以通过其他文章和更深入的模型展开进一步探索。

## 8.2　文本摘要

文本摘要是将输入文档转换为简短摘要的过程，能帮助我们在短时间内理解文档的主要内容。从根本上讲，有两种类型的摘要：一种是提取式摘要，另一种是抽象式摘要。我们将简要介绍这些摘要类型。

### 8.2.1　提取式摘要

在提取式摘要中，文档中的重要短语或关键字会被提取并连接起来以获得简短的摘要。

这种摘要的主要优点在于简单、稳健，因为所提取的文本直接来自文档。但该方法的缺点是我们可能无法获得新的措辞从而使摘要更为清晰。接下来，我们将简要介绍使用 gensim 的提取式摘要。

#### 使用 gensim 进行摘要

gensim 的摘要算法是基于 Rada Mihalcea 等人所创 TextRank 算法的改进版本。TextRank 是一种基于图的算法，使用文档中的关键字作为顶点，而关键字之间边缘的权重是根据其在文本中的共现来确定的。TextRank 与 PageRank 类似，能用于确定关键字的重要性。最后，它通过对包含高排名关键字的重要句子进行排名来提取摘要。根据该描述可知，TextRank 很明显是提取摘要

器的一个示例。我们将查看一个使用 gensim 摘要生成器的简单示例，并使用 nltk 产品评论语料库作为测试数据：

```
from nltk.corpus import product_reviews_1
from gensim.summarization import summarizer
```

我们也导入了 gensim 摘要器，之后将使用它来生成产品评论摘要：

```
product_review_raw =
product_reviews_1.raw('Apex_AD2600_Progressive_scan_DVD player.txt')
product_summary = summarizer.summarize(product_review_raw,word_count=100)
print("Raw Text Length: ", len(product_review_raw.split()))
print("Summary Length: ", len(product_summary.split()))
print("Summary: ", product_summary)
```

我们选择了 nltk 语料库中的一种产品（DVD 播放器），并通过 `summary` 函数的 `word_count` 参数将摘要限制为 100 个单词。我们将打印原始文本和摘要文本的单词长度，以查看两者之间的区别：

```
Raw Text Length: 13014
Summary Length: 88
Summary: player[+2]##i bought this apex 2600 dvd player for myself at
christmas because it got good reviews as a good value for the money on a
variety of different sites . remote[-2]##we 've purchased 3 universal
remotes so far-all claiming to work " apex " dvd players and none worked .
##after having bought and been disappointed in another brand of dvd player
, i purchased the apex ad2600 from amazon and first of all i should say it
was delivered much more quickly than i had expected .
```

输出显示，摘要文本大约有 88 个单词，而原始产品评论有 13 014 个单词。我们还可以查看摘要器提取的关键字：



```
from gensim.summarization import keywords
keywords(product_review_raw).split("\n")[0:20]
```

`keyword` 模块将提取文档中的主要关键字，其中前 20 个关键字如下所示：

```
['players',
 'dvd player',
 'dvds',
 'play',
 'playing',
 'plays',
 'apex',
 'picture',
 'pictures',
 'pictured',
 'remotes',
 'work',
 'works',
 'working',
```

```
'worked',
'customer',
'customers',
'disks types played',
'problems',
'problem']
```

该输出还显示出了某些关键字属于同一单词的不同时态。

接下来，我们将使用深度学习来研究抽象式摘要器。

### 8.2.2　抽象式摘要

抽象式摘要生成的输出摘要包含不在原始文本中的单词或短语，同时保留输入文档的原始意图。这种摘要方法可以产生全新的短语，从而得出更为自然的摘要。我们将首先看一下使用深度学习方法的抽象文本摘要。在文本摘要中，输入和输出都是文本序列。实践中常用的深度学习模型是序列到序列（Seq2Seq）模型。接下来，我们将简要描述这种文本摘要方法。

#### 1. 编码器–解码器架构

顾名思义，编码器–解码器架构由一个编码器和一个解码器组件组成，图 8-3 对此进行了说明。**编码器**的功能是获取输入文本序列并将其转换为密集向量表示，也称为**thought vector**[①]或**上下文向量**（context vector）。本质上，thought vector是一种可捕获整个输入文本的上下文含义的内部表示形式。**解码器**采用完整原始文本的密集向量表示，并生成输出摘要，一次只生成一个单词。

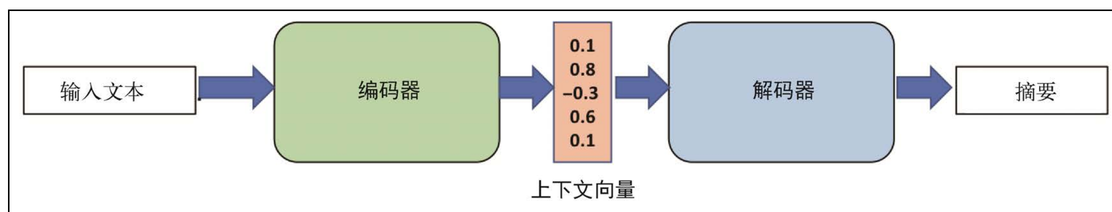



图　8-3

#### 2. 编码器

最常见的编码器类型使用双向 RNN，且带有 LSTM 或 GRU 单元。在这种情况下，编码器的输入是单词或词嵌入的分布式表示。

#### 3. 解码器

解码器是另一个双向 LSTM 或 GRU 网络。它采用编码器发出的输入文本和先前所生成摘要

① thought vector 是 Google 著名的深度学习研究人员 Geoffrey Hinton 推广的术语，它使用基于自然语言的向量来改善搜索结果。——译者注

字的向量表示来生成下一个摘要字。

#### 4. 使用 GRU 进行新闻摘要

在本例中，我们将研究如何使用 LSTM 对新闻文章进行摘要。本例的完整 Jupyter Notebook 可在本书的代码库中找到（Chapter08/02_example.ipynb）。除了前面描述的基本架构之外，我们还将使用附加的 attention 层。从之前文本摘要的研究工作可以确定，具有 attention 机制的模型要优于没有注意力模型的模型。

#### 5. 数据准备

首先，导入原始新闻数据和摘要到 pandas DataFrame：

```
titledata=[]
artdata=[]
with gzip.open('data/news.txt.gz') as artfile:
    for li in artfile:
        artdata.append(li)
with gzip.open('data/summary.txt.gz') as titlefile:
    for li in titlefile:
        titledata.append(li)
news = pd.DataFrame({'Text':artdata,'Summary':titledata})
news = news.sample(frac=0.1)
news['Text_len'] = news.Text.apply(lambda x: len(x.split()))
news['Summary_len'] = news.Summary.apply(lambda x: len(x.split()))
```

我们将查看示例新闻 `Text` 和 `Summary`：

```
print(news['Text'].head(2).values)
print(news['Summary'].head(2).values)

Output:
[b'chinese president hu jintao said here monday that china will work with
romania to promote bilateral trade and economic cooperation .\n'  b'federal
reserve policymakers opened a two-day meeting here tuesday to debate us
monetary moves , a fed source reported .\n']

[b'chinese president meets romanian pm\n'  b'federal reserve policymakers
open two-day meeting\n']
```

对于词嵌入，我们将使用 `glove.6B` 向量语料库并将其载入到嵌入矩阵中去。

```
def build_word_vector_matrix(vector_file):
    embedding_index = {}
    with codecs.open(vector_file, 'r', 'utf-8') as f:
        for i, line in enumerate(f):
            sr = line.split()
            if(len(sr)&lt;26):
                continue
    word = sr[0]
    embedding = np.asarray(sr[1:], dtype='float32')
```

```
        embedding_index[word] = embedding
        return embedding_index

    embeddings_index =
      build_word_vector_matrix('/Users/i346047/prs/temp/glove.6B.50d.txt')
```

embeddings_index 包含单词到对应词向量的映射。接下来，我们将为新闻文本和摘要中的所有单词创建从单词到整数索引的映射（反之亦然）：

```
word2int = {}
count_threshold = 20
value = 0
for word, count in word_counts_dict.items():
    if count &gt;= count_threshold or word in embeddings_index:
        word2int[word] = value
        value += 1


special_codes = [TOKEN_UNK,TOKEN_PAD,TOKEN_EOS,TOKEN_GO]

for code in special_codes:
    word2int[code] = len(word2int)

int2word = {}
for word, value in word2int.items():
    int2word[value] = word
```

请注意，我们还针对词汇表中不存在的单词（UNK）、填充标记（PAD）、句子结尾（EOS）和起始标记（GO）纳入了特殊代码，由代码中的相应常量定义。填充标记将句子填充到新闻文本和摘要中的固定长度。在训练和推理过程中，句子标记的开头和结尾分别作为前缀和后缀，附加到输入的新闻文本中。接下来，将新闻文本和摘要转换为整数 ID：

```
def convert_sentence_to_ids(text, eos=False):
    wordints = []
    word_count = 0
    for sentence in text:
        sentence2ints = []
    for word in sentence.split():
        word_count += 1
        if word in word2int:
            sentence2ints.append(word2int[word])
        else:
            sentence2ints.append(word2int[TOKEN_UNK])
    if eos:
        sentence2ints.append(word2int[TOKEN_EOS])
    wordints.append(sentence2ints)
return wordints, word_count
```

请注意，对于不在词汇表中的单词，我们分配了 UNK 标记 ID。与之类似，我们以 EOS 标记来结束句子。接下来，我们将丢弃所有不在指定最小和最大句子长度内的输入文本和摘要，还将删除一些包含大于限制的未知单词的句子：

```
news_summaries_filtered = []
news_texts_filtered = []
max_text_length = int(news.Text_len.mean() + news.Text_len.std())
max_summary_length = int(int(news.Summary_len.mean() +
news.Summary_len.std()))
min_length = 4
unknown_token_text_limit = 10
unknown_token_summary_limit = 4
for count,text in enumerate(id_texts):
    unknown_token_text = unknown_tokens(id_texts[count])
    unknown_token_summary = unknown_tokens(id_summaries[count])
    text_len = len(id_texts[count])
    summary_len = len(id_summaries[count])
    if((unknown_token_text&gt;unknown_token_text_limit) or
(unknown_token_summary&gt;unknown_token_summary_limit)):
     continue
     if(text_len&lt;min_length or summary_len&lt;min_length or
text_len&gt;max_text_length or      summary_len&gt;max_summary_length):
     continue
     news_summaries_filtered.append(id_summaries[count])
     news_texts_filtered.append(id_texts[count])
```

我们用输入文本和摘要平均长度的一个标准偏差设定了最小和最大长度，你可以通过更改变量 max_summary_length、max_text_length、unknown_token_text_limit、unknown_token_summary_limit 和 min_length 做出修改。

现在我们来看一看模型的创建。

### 6. 编码网络

对于编码器组件，我们利用带有 GRU 单元的双向 RNN。当然也可以使用 LSTM 来代替 RNN，你可以对此展开尝试以观察模型性能的差异：



```
def get_cell(csize,dprob):
    rnc = GRUCell(csize)
    rnc = DropoutWrapper(rnc, input_keep_prob = dprob)
    return rnc

def encoding_layer(csize, len_s, nl, rinp, dprob):
    for l in range(nl):
        with tf.variable_scope('encoding_l_{}'.format(l)):
            rnn_frnt = get_cell(csize,dprob)
            rnn_bkwd = get_cell(csize,dprob)
            eop, est = tf.nn.bidirectional_dynamic_rnn(rnn_frnt, rnn_bkwd,
                                                        rinp,
                                                        len_s,
dtype=tf.float32)
    eop = tf.concat(eop,2)
    return eop, est
```




请注意，因为这是双向的，所以我们串联了编码器的输出。

### 7. 解码网络

像编码器一样，对于解码器网络，我们将使用带有 GRU 单元的 RNN。我们还将使用 `tf.contrib.rnn.DropoutWrapper` 封装器类，通过 `dropout` 创建 `nlyrs` 个 GRU 层，并利用 `BahdanauAttention` 机制将注意力集中在编码器的输出上：

```
def decoding_layer(dec_emb_op, embs, enc_op, enc_st, v_size, txt_len,
                   summ_len,mx_summ_len, rnsize, word2int, dprob,
batch_size, nlyrs):
    for l in range(nlyrs):
        with tf.variable_scope('dec_rnn_layer_{}'.format(l)):
            gru = tf.contrib.rnn.GRUCell(rnn_len)
            cell_dec = tf.contrib.rnn.DropoutWrapper(gru,input_keep_prob =
dprob)
    out_l = Dense(v_size, kernel_initializer =
tf.truncated_normal_initializer(mean = 0.0, stddev=0.1))
    attention = BahdanauAttention(rnsize, enc_op,txt_len,
                                                  normalize=False,
                                                  name='BahdanauAttention')
    cell_dec = AttentionWrapper(cell_dec,attention,rnn_len)
    attn_zstate = cell_dec.zero_state(batch_size , tf.float32 )
    attn_zstate = attn_zstate.clone(cell_state = enc_st[0])
    with tf.variable_scope("decoding_layer"):
        tr_dec_op = trng_dec_layer(dec_emb_op,
                                                  summ_len,
                                                  cell_dec,
                                                  attn_zstate,
                                                  out_l,
                                                  v_size,
                                                  mx_summ_len)
    with tf.variable_scope("decoding_layer", reuse=True):
        inf_dec_op = infr_dec_layer(embs,
                                                    word2int[TOKEN_GO],
                                                    word2int[TOKEN_EOS],
                                                    cell_dec,
                                                    attn_zstate,
                                                    out_l,
                                                    mx_summ_len,
                                                    batch_size)
    return tr_dec_op, inf_dec_op
```

为了生成 attention 和序列到序列，我们将使用 TensorFlow 中 seq2seq 库中的类。现在来看看训练期间的解码是如何执行的：

```
def trng_dec_layer(dec_emb_inp, summ_len, cell_dec, st_init, lyr_op,
                            v_size, max_summ_len):
    helper = TrainingHelper(inputs=dec_emb_inp,sequence_length=summ_len,
time_major=False)
    dec = BasicDecoder(cell_dec,helper,st_init,lyr_op)
    logits, _, _ =
dynamic_decode(dec,output_time_major=False,impute_finished=True,
                                  maximum_iterations=max_summ_len)
    return logits
```

我们使用 tf.contrib.seq2seq.TrainingHelper 类将真实摘要提供给解码器输入以进行训练。attention 状态也被通过 initial_state 张量作为输入传递给了解码器。最后，tf.contrib.seq2seq.dynamic_decode 函数对此输入执行了解码。

### 8. 序列到序列

接下来，我们将研究将所有这些进行封装的序列到序列高级函数，该函数读取输入文本句子的词嵌入，创建编码/解码层，并生成 logits 作为输出。op_tr 和 op_inf 对象分别表示训练和推理期间的预测：

```
def seq2seq_model(data_inp, data_summ_tgt, dprob, len_txt, len_summ,
max_len_summ,
                  v_size, rnsize, nlyrs, word2int, batch_size):
    inp_emb = word_emb_matrix
    word_embs = tf.Variable(inp_emb, name="word_embs")
    inp_enc_emb = tf.nn.embedding_lookup(word_embs, data_inp)
    op_enc, st_enc = encoding_layer(rnsize, len_txt, nlyrs, inp_enc_emb,
dprob)
    inp_dec = process_encoding_input(data_summ_tgt, word2int, batch_size)
    inp_dec_emb = tf.nn.embedding_lookup(inp_emb, inp_dec)
    op_tr, op_inf = decoding_layer(inp_dec_emb,
                                                        inp_emb,
                                                        op_enc,
                                                        st_enc,
                                                        v_size,
                                                        len_txt,
                                                        len_summ,
                                                        max_len_summ,
                                                        rnsize,
                                                        word2int,
                                                        dprob,
                                                        batch_size,
                                                        nlyrs)
    return op_tr, op_inf
```

输出训练日志 op_tr（以及真实摘要）用于计算训练期间的代价变化。

现在我们看看如何建立图。

### 9. 建立图

我们使用高级 seq2seq_model 函数构建图。以下是建立图和优化器的代码：

```
train_graph = tf.Graph()
with train_graph.as_default():
    data_inp, tgts, lrt, dprobs, len_summ, max_len_summ, len_txt =
model_inputs()

    tr_op, inf_op = seq2seq_model(tf.reverse(data_inp, [-1]),
                                                      tgts,
                                                      dprobs,
                                                      len_txt,
```

```
                                                          len_summ,
                                                          max_len_summ,
                                                          len(word2int)+1,
                                                          rnn_len,
                                                          n_layers,
                                                          word2int,
                                                          batch_size)
    tr_op = tf.identity(tr_op.rnn_output, 'tr_op')
    inf_op = tf.identity(inf_op.sample_id, name='predictions')
    seq_masks = tf.sequence_mask(len_summ, max_len_summ, dtype=tf.float32,
name='masks')

    with tf.name_scope("optimizer"):
        tr_cost = sequence_loss(tr_op,tgts,seq_masks)
        optzr = tf.train.AdamOptimizer(lrt)
        grds = optzr.compute_gradients(tr_cost)
        capped_grds = [(tf.clip_by_value(grd, -5., 5.), var) for grd, var
in grds
                        if grd is not None]
        train_op = optzr.apply_gradients(capped_grds)
    tf.summary.scalar("cost", tr_cost)
print("Graph created.")
```

我们利用 `AdamOptimizer` 最小化通过训练 logits（逻辑模型）和目标摘要词所计算出的代价。请注意，因为我们使用的是 RNN，所以要为梯度设置上限，以避免出现梯度爆炸的问题。

#### 10. 训练

在训练中，我们将使用一部分输入数据，也可以通过增加输入数据以改善模型的性能：

```
min_learning_rate = 0.0006
display_step = 20
early_stop_cnt = 0
early_stop_cnt_max = 3
per_epoch = 3


update_loss = 0
batch_loss = 0
summary_update_loss = []

news_summaries_train = news_summaries_filtered[0:3000]
news_texts_train = news_texts_filtered[0:3000]
update_check = (len(news_texts_train)//batch_size//per_epoch)-1
checkpoint = logs_path + 'best_so_far_model.ckpt'
with tf.Session(graph=train_graph) as sess:
    tf_summary_writer = tf.summary.FileWriter(logs_path, graph=train_graph)
    merged_summary_op = tf.summary.merge_all()
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(1, epochs+1):
        update_loss = 0
        batch_loss = 0
        for batch_i, (summaries_batch, texts_batch, summaries_len,
texts_len) in enumerate(
                get_batches(news_summaries_train, news_texts_train,
```

```
batch_size)):
            before = time.time()
            _,loss,summary = sess.run(
                [train_op, tr_cost,merged_summary_op],
                {data_inp: texts_batch,
                 tgts: summaries_batch,
                 lrt: lr,
                 len_summ: summaries_len,
                 len_txt: texts_len,
                 dprobs: dr_prob})
```

我们可以从 get_batches 函数获取用于训练的批数据。该函数通过填充开始和结束标记来格式化输入文本和摘要。你可以更改参数，例如学习率、层数和 dropout 概率来试验模型的性能，也可以增加训练周期以提高模型的准确率。只要当前批次损失较先前有所改善，模型也会被保存下来。

现在可以看看模型的最终输出：

```
No Improvement.
** Epoch  20/20 Batch   20/1048 - Batch Loss:   1.454, seconds: 16.15
Average loss: 1.383
Saving model
** Epoch  20/20 Batch   40/1048 - Batch Loss:   1.362, seconds: 17.11
Average loss: 1.353
```

可以看到，损失从初始的 9.254 降到了大概 1.353。请注意，每次运行可能都有区别。

#### 11. 推理

现在，我们将加载已保存的模型文件，并使用模型未知的输入文本样本数据对其进行测试。我们将从模型中加载 logits 张量，并使用它来推理未知数据或测试数据的摘要文本以预测输出：

```
INFO:tensorflow:Restoring parameters from
/tmp/models/best_so_far_model.ckpt

Text
Word Ids:   [1286, 4050, 1283, 30463, 1284, 30648, 538, 173, 1286, 1287, 166, 82, 2122,
8272, 1289, 179, 1290, 6526, 2122, 2518, 976, 24, 6001, 30427, 2360, 162, 20415, 977,
30428, 54118]

Input Words: space shuttle atlantis ' astronauts thanked the international space
station residents for their warm hospitality and then began their journey home to earth ,
ending a weeklong visit .

Summary
Word Ids:       [1283, 1284, 1285, 1286, 1286, 1286]
Response Words: atlantis astronauts thank space space space
Ground Truth: atlantis astronauts thank space station men for hospitality then
```

我们还将对训练数据进行推理。尽管未知输入文本的预测或摘要与实际情况相去甚远，但对于训练数据的预测却非常接近，如以下输出所示：

8

```
INFO:tensorflow:Restoring parameters from
/tmp/models/best_so_far_model.ckpt

Text
Word Ids:    [10651, 1190, 910, 2324, 457, 178, 1203, 3909, 126, 7746, 909,
910, 29, 33, 4642, 745, 10903, 581, 13, 33, 911, 2796, 19, 2156, 115, 526,
11024, 1871, 10589, 5672, 702, 10590, 11026]

Input Words: japanese share prices closed #.## percent higher thursday as
easing oil prices and a weaker yen buoyed confidence in a market continuing
to recover from the shock , dealers said .

Summary
Word Ids:       [548, 492, 493, 457, 178, 457]
Response Words: tokyo shares close #.## percent #.##
Ground Truth:   tokyo stocks close up #.## percent
```

应当注意的是，模型在测试数据上性能较差的原因可以归结于我们用于训练的数据集过小（训练大小为 3000 个样本）。你可以使用相同的模型在更大的数据集上（可能在 GPU 上）进行训练，以获得更好的结果。

### 12. TensorBoard 可视化

现在，我们将使用 TensorBoard 简要查看图和损失函数。首先来看看 TensorBoard 的图输出，如图 8-4 所示。

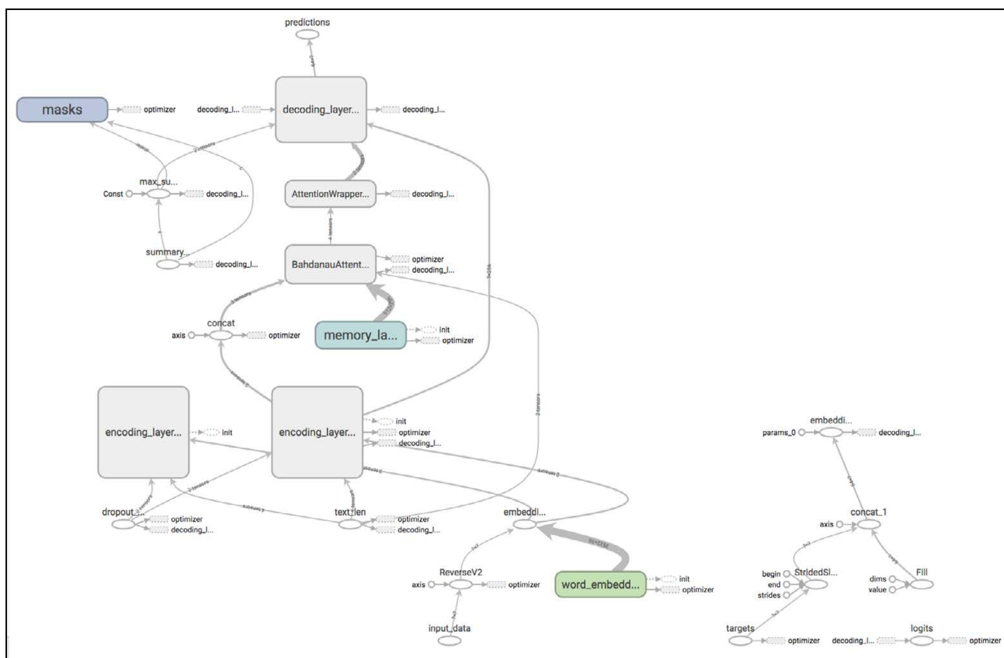



图 8-4　TensorBoard 图输出

可以清楚地看到，编码层和解码层构成了模型的主要部分，而 Bahdanau attention 机制则是解码层中的另一个输入。首先将输入的嵌入内容馈送到编码层，将其输出馈送到 attention 机制以及解码层。最后，解码层提供预测作为输出。优化器接受解码层输出、attention 机制输出和目标以对代价函数进行优化，如图 8-5 所示。

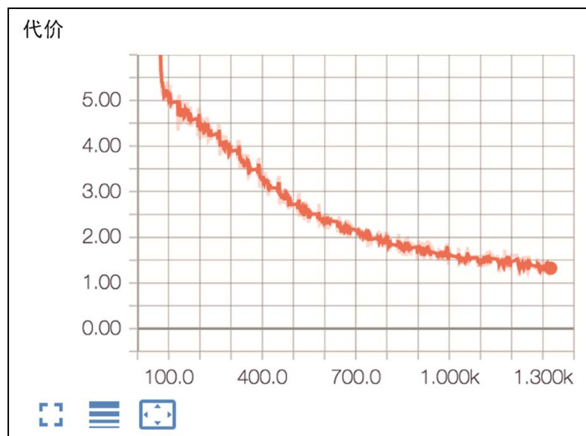



图 8-5

我们还发现，代价随着训练步数的增多而稳步下降。在下一节中，我们将查看一些近期的论文，其中描述了编码器–解码器模型（带有 attention 机制）的增强功能。

## 8.2.3 最新抽象式文本摘要

在本节中，我们将看到两篇近期的论文，它们描述了上一节新闻文本摘要示例中所使用的模型的增强功能。

第一篇论文是“Abstractive Text Summarization Using Sequence-To-Sequence RNNs and Beyond”，作者是来自 IBM 的 Ramesh Nallapati 等人。他们将神经机器翻译模型应用于文本摘要，并且与最新的系统相比获得了更好的性能。该模型使用双向 GRU-RNN 作为编码器，并使用单向 GRU-RNN 作为解码器。请注意，这与新闻摘要示例中使用的模型架构相同。

以下是他们提出的主要附加增强。

- 除了词嵌入之外，还增强了输入特征，包括词性标签、命名实体标签和单词的 TF-IDF 统计信息。这有助于识别文档中的关键概念和实体，从而改进摘要文本。
- 这些附加特征与现有的词向量串联在一起被馈入编码器。

图 8-6 说明了附加使用的特征：词嵌入（W）、词性（POS）、命名实体标签（NER）以及术语频率、逆文档频率（TF-IDF）。

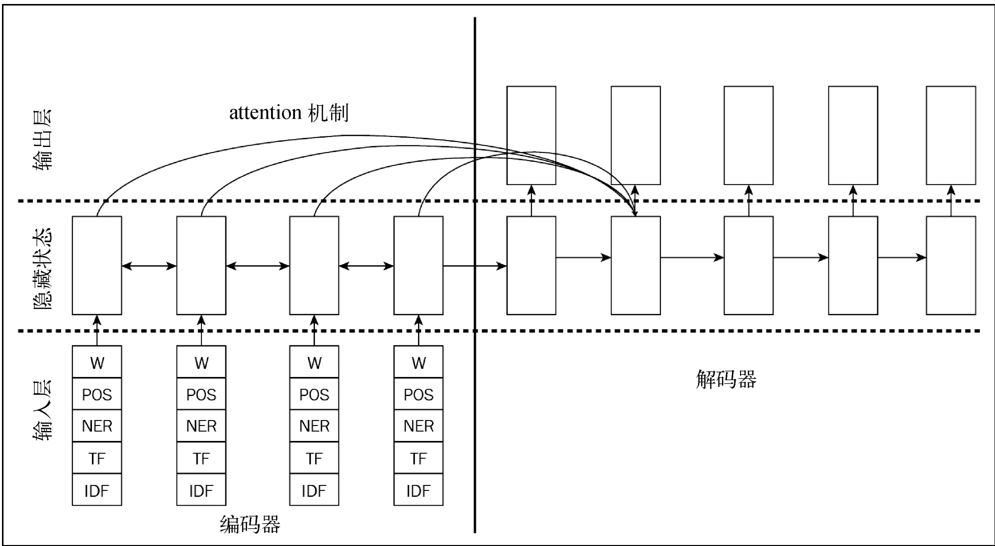



图　8-6

为了处理测试数据中未知单词或外来（out-of-vocabulary，OOV）单词，他们使用了生成器指针网络，该网络在激活时从源文档位置复制单词。我们前面描述的新闻摘要示例只是简单地忽略了 OOV 单词，并用 `UNK` 标记进行了替换，这可能不会生成连贯的摘要。

包含指针生成器网络的架构如图 8-7 所示。

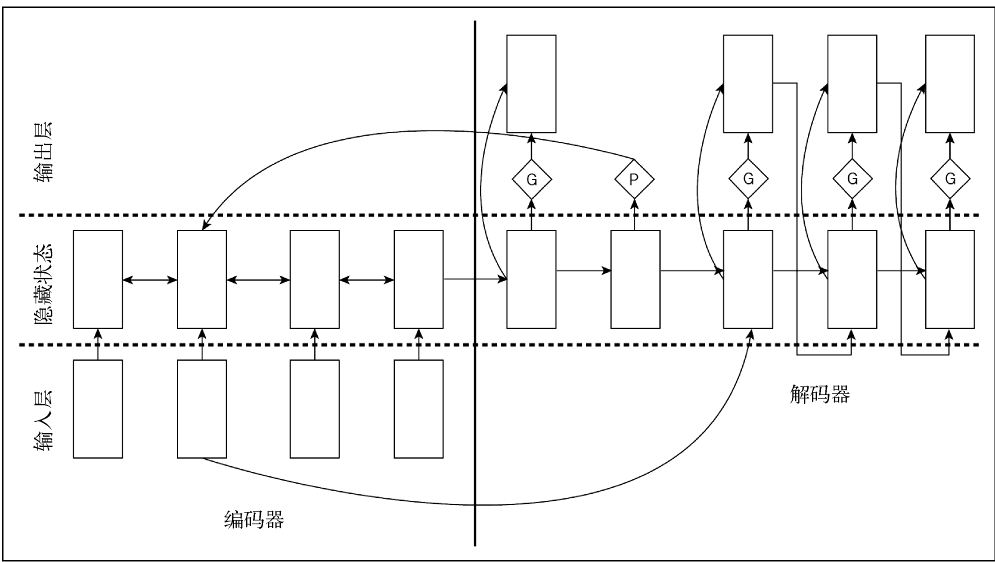



图　8-7

对于文档较长、句子很多的源数据集，还有必要确定用于摘要的关键句子。为此，他们在编码侧使用了带有两个双向 RNN 的分层网络：一个作用于单词级别，另一个则作用于句子级别。

类似于 Ramesh Nallapati 等人采用的方法，来自 Google 的 Abigail See 等人的论文“Get to the Point: Summarization with Pointer-Generator Networks”也使用了指针生成器网络。但是，除了只处理 OOV 单词外，他们还考虑了副本分发和词汇分布。副本分发要考虑在源文档中被重复使用的特定单词及其 attention，这增加了在摘要中选择单词的可能性。图 8-8 显示了如何组合词汇分配和 attention 分配以生成最终摘要。

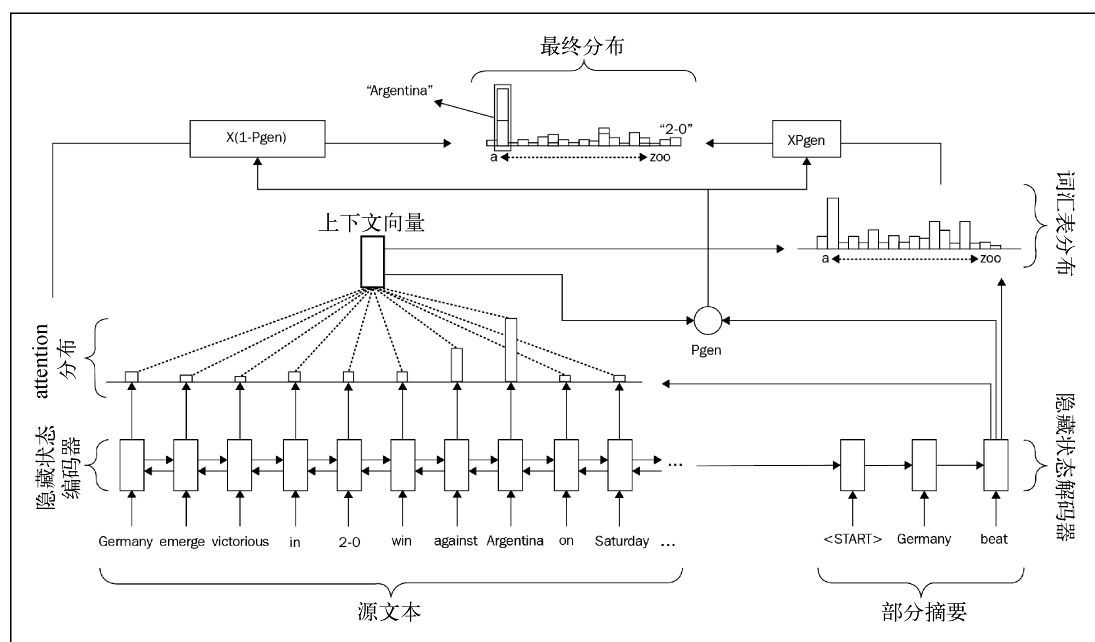



图 8-8

## 8.3 小结

本章重点介绍了文本生成和文本摘要。我们使用 GRU 和 RNN 演示了一个文本生成模型示例来生成 Linux 内核代码。将这些模型应用于不同的领域或源输入文本，可以帮助我们理解其基础结构和上下文。接下来，我们描述了文本摘要的不同类型，介绍了一种简单的提取式摘要方法，并使用 gensim 生成产品评论摘要。提取式摘要可从源文本复制单词，而抽象式摘要可生成全新而直观的摘要。

为了全面介绍抽象式摘要，我们引入了一种编码器–解码器模型，该模型使用 GRU 和 RNN 来摘要新闻文本。最后，我们研究了一些先进方法以改进基本编码器–解码器模型（带有 attention 机制）。你可以在我们开发的基础模型上构建，以纳入这些增强功能。一种简单的增强是将附加特征（例如 POS、NER 和 TF-IDF）添加到输入文本的词嵌入中。

在下一章，我们将讨论问答和聊天机器人这个有趣的主题。我们将开发一个问答模型，并使用生成 RNN 模型构建一个聊天机器人。

第 9 章

# 使用记忆网络完成问答任务和编写聊天机器人

**自然语言理解**（natural language understanding，NLU）任务可被视为一个概括性术语，它涵盖的研究领域涉及以句法和语义两种方式对文本进行推理，例如文本摘要、机器翻译和对话建模（即聊天机器人）。

NLP 的一个有趣的研究方向是将所有 NLU 任务分解为一个简单的**问答**（question-answer，QA）框架。在该框架中，模型必须根据输入文本（以有关狗的 Wikipedia 文章为例）进行推理并回答问题：最常见的犬种是什么？文章摘要是什么？摘要的法语翻译又是什么？

在本章中，我们将了解 QA 任务，并介绍一类称为**记忆网络**的深度学习模型以构建 QA 系统。然后，我们将了解端到端训练聊天机器人模型的各个组成部分，并扩展记忆网络以构建会话式聊天机器人。

## 9.1 QA 任务



从表面上来看，QA 任务似乎很简单——给定一个问题和一些相关事实（可选），模型要产生一个答案。

传统的 QA 方法包括基于规则的模型以及基于单词重叠或 TF-IDF 分数的信息检索方法。然而，由于自然语言的内在复杂性，训练模型以在语法和语义上理解输入以及相关事实还是有挑战性的。深度神经网络无须手工或特征工程就能学会对复杂信息的建模，已逐渐成为处理这些任务的先进网络。

### QA数据集

根据是否要求回答或答案的形式，各个问答数据集有所不同。我们将简要概述一些常见的 QA 学术数据集及其主要特征，如表 9-1 所示。

表 9-1

| 数据集名称 | 描 述 | 类 别 |
|---|---|---|
| bAbI text understanding tasks | 这套包含 20 个综合生成任务的套件旨在测试 NLU 模型所应具备的一些基本技能。每个任务都根据在其环境中采取各种动作的段落来训练模型，以回答有关该环境状态的问题 | 选择 |
| SQuAD: Stanford Question Answering Dataset | SQuAD 包含与 Wikipedia 文章相关的问题，并且要求模型在文章中选择答案范围作为对该问题的答案。它是当今最受欢迎的 QA 据集 | 答案范围 |
| VQA: Visual Question Answering Dataset | 在 VQA 中，要进行推理的输入是图像而非文本。模型必须学会对像素进行推理以选择有关图像文本问题的答案 | 选择 |
| AI2 Reasoning Challenge | ARC 数据集包含科学多选题，以从中选择答案。它是专门为揭示当前神经网络模型的缺点而设计的（这些模型声称可以对如 SQuAD 和 bAbI 的简单数据集进行语言理解） | 多选 |

## 9.2 用于 QA 任务的记忆网络

2014 年，Weston 等人在端到端训练的 QA 系统的基础上提出了记忆网络，这是用于 NLU 任务的一类神经网络模型。给定问题和一些支持事实或相关信息，其任务是生成或选择适当的答案。该模型将这些事实存储在持久性内存中，并被训练以根据这些事实执行推理，产生适当的响应。



*有关该主题的第一篇论文名为“Memory Networks”，作者是 Jason Weston、Sumit Chopra 和 Antoine Bordes。*

QA 任务种类繁多，因此记忆网络提供了一种灵活的模块化框架，存储在内存中的事实可能从文本到图像不等，并且答案可以从一组候选项中生成或检索出来。

### 9.2.1 记忆网络管道概述

记忆网络的架构通常可以分解为四个部分：**问题模块**、**输入模块**、**记忆模块**和**输出模块**。按照神经网络的惯例，信息通过密集的向量/嵌入从一个模块传递到另一个模块，从而使用梯度下降来端对端地训练模型的参数，如图 9-1 所示。

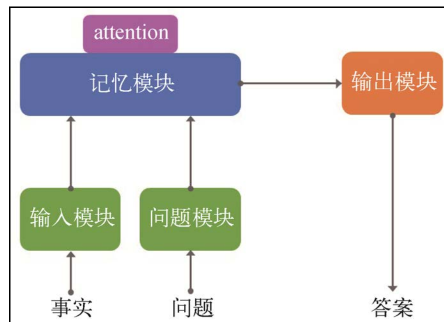



图 9-1

下面解释了模型的工作方式。

- **输入模块**收到多个事实并将其编码到向量中。
- 与**输入模块**类似，**问题模块**也负责将问题编码为向量。
- **记忆模块**从**输入模块**接收编码好的事实，从**问题模块**接收编码好的问题，并对事实执行 soft attention 机制以弄清其与问题之间的相关性。attention 的结果是给定问题的上下文向量，对问题以及回答该问题所需的所有上下文信息进行编码。
- **输出模块**接收上下文向量，并负责产生对应格式的答案。这可能意味着从候选集中选择适当的响应、对答案范围的预测或逐词生成响应。

## 9.2.2　使用TensorFlow写一个记忆网络

在以下小节中，我们将研究 Sukhbaatar 等人提出的简单记忆网络架构（他们使用该网络在 2015 年建立了基于检索的 QA 系统）。我们将提供代码片段以构建通用的记忆网络类，并在此过程中说明模型的工作原理和详细信息。



关于该模型的详细信息可以在 Sainbayar Sukhbaatar、Arthur Szlam、Jason Weston 和 Rob Fergus 的论文“End-to-End Memory Networks”中找到。

### 1. 类构造器

我们将定义一个构造函数，用于初始化记忆网络的 `initializer` 对象、`optimizer` 对象和最小批大小等参数。我们还将为损失、预测和训练编写高级 TensorFlow 操作。所有这些都取决于`_inference` 方法，我们将在下面对该方法进行实现：

```
class MemoryNetwork(object):
    def __init__(self, sentence_size, vocab_size, candidates_size,
                 candidates_vec, embedding_size, hops,
                 initializer=tf.random_normal_initializer(stddev=0.1),
                 optimizer=tf.train.AdamOptimizer(learning_rate=0.01),
                 session=tf.Session()):
        self._hops = hops
        self._candidates_vec = candidates_vec
        # 定义模型的输入占位符
        self._facts = tf.placeholder(
            tf.int32, [None, None, sentence_size], name="facts")
        self._questions = tf.placeholder(
            tf.int32, [None, sentence_size], name="questions")
        self._answers = tf.placeholder(
            tf.int32, [None], name="answers")

        # 定义用于推理的训练变量
        with tf.variable_scope("MemoryNetwork"):
            # 用于输入事实和问题的嵌入查找矩阵
            self.word_emb_matrix = tf.Variable(initializer(
```

```
            [vocab_size, embedding_size]), name="A")
        # 在推理中用于线性变换的矩阵
        self.transformation_matrix = tf.Variable(initializer(
            [embedding_size, embedding_size]), name="H")
        # 用于输出响应的词嵌入
        self.output_word_emb_matrix = tf.Variable(initializer(
            [vocab_size, embedding_size]), name="W")

    # 在推理预测上计算交叉熵误差
    logits = self._inference(self._facts, self._questions)
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits=logits, labels=self._answers, name="cross_entropy")
    cross_entropy_sum = tf.reduce_sum(
        cross_entropy, name="cross_entropy_sum")

    # 定义损失操作
    self.loss_op = cross_entropy_sum

    # 定义梯度管道
    grads_and_vars = optimizer.compute_gradients(self.loss_op)
    # 定义训练操作
    self.train_op = optimizer.apply_gradients(
        grads_and_vars, name="train_op")

    # 定义预测操作
    self.predict_op = tf.argmax(logits, 1, name="predict_op")

    # 加载会话并初始化所有变量
    self._session = session
    self._session.run(tf.initialize_all_variables())
```

### 2. 输入模块

输入模块对每个输入事实中的所有单词进行词嵌入查找，然后沿时序方向求和（即对事实中每个单词的嵌入求和）来为每个事实构建单独嵌入：

```
def _input_module(self, facts):
    with tf.variable_scope("InputModule"):
        facts_emb = tf.nn.embedding_lookup(self.word_emb_matrix,
                                           facts)
        return tf.reduce_sum(facts_emb, 2)
```

### 3. 问题模块

问题模块执行与输入模块相同的嵌入查找和时间求和任务。两个模块之间共享嵌入矩阵和单词词汇：

```
def _question_module(self, questions):
    with tf.variable_scope("QuestionModule"):
        questions_emb = tf.nn.embedding_lookup(
            self.word_emb_matrix, questions)
        return tf.reduce_sum(questions_emb, 1)
```

由于我们正在构建概念上最为简单的记忆网络，因此不使用复杂的句子表示模型，例如 RNN 或 CNN。记忆网络架构的模块化性质让进一步开展实验变得非常轻松。

### 4. 记忆模块

记忆网络模型的神奇之处在于记忆模块，该模块对事实嵌入执行 soft attention 机制。有关记忆网络和其他基于 attention 的模型的文献介绍了许多类型的 attention 机制，但是这些机制都依赖于元素点积的概念，并且计算两个向量之间的和以作为衡量语义或句法相似性的运算。我们称其为 reduce-dot 操作，该操作接收两个向量并得到一个表示相似度分数的数。

我们制定了以下 attention 机制。

(1) 上下文向量用于对产生输出所需的所有信息进行编码，并被初始化为问题向量。

(2) 每个事实向量和上下文向量之间的 reduce-dot 操作为我们提供了每个事实向量的相似性分数。

(3) 然后，对这些相似性分数进行 softmax 运算，将其归一化为 0 和 1 之间的概率值。

(4) 对于每个事实向量，都将向量的每个元素乘以其相似度概率值。

(5) 最后，对这些加权事实向量进行元素求和，以获取上下文表示，其中某些事实比其他事实具有更高的权重。

(6) 通过在元素上添加此上下文表示来更新上下文向量。

(7) 更新的上下文向量用于关注事实向量，随后使用事实的多次遍历（称为**跃点**）来进一步更新。

这些步骤可以通过记忆模块的拓展视角进行理解，如图 9-2 所示。

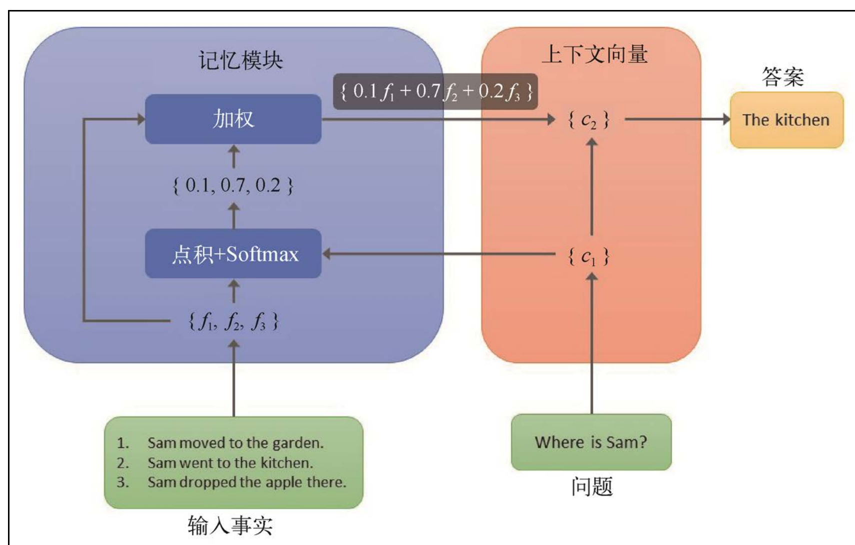



图　9-2

因为没有用于执行 reduce-dot 操作的高级封装 API，所以必须在 TensorFlow 中使用类似于 NumPy 的原子操作来编写我们的记忆模块：

```
def _memory_module(self, questions_emb, facts_emb):
    with tf.variable_scope("MemoryModule"):
        initial_context_vector = questions_emb
        context_vectors = [initial_context_vector]
        # 在事实上执行多跃点 attention 以更新上下文向量
        for hop in range(self._hops):
            # 执行 reduce_dot
            context_temp = tf.transpose(
                tf.expand_dims(context_vectors[-1], -1), [0, 2, 1])
            similarity_scores = tf.reduce_sum(
                facts_emb * context_temp, 2)
            # 计算相似度概率
            probs = tf.nn.softmax(similarity_scores)
            # 执行 attention 乘法
            probs_temp = tf.transpose(tf.expand_dims(probs, -1),
                                      [0, 2, 1])
            facts_temp = tf.transpose(facts_emb, [0, 2, 1])
            context_rep = tf.reduce_sum(facts_temp*probs_temp, 2)
            # 更新上下文向量
            context_vector = tf.matmul(context_vectors[-1],
                                       self.transformation_matrix) \
                                 + context_rep
            # 添加到上下文向量列表末尾以在下个跃点使用
            context_vectors.append(context_vector)
        # 返回最后一个跃点的上下文向量
        return context_vector
```

每个跃点可能会关注事实的不同方面。使用这种多跃点 attention 机制会产生更为丰富的上下文向量。该机制使模型能够逐步了解事实并进行推理，通过可视化每个跃点的相似度概率值可以看到，如图 9-3 所示。

| 故事（2：2支撑事实） | 跃点1 | 跃点2 | 跃点3 |
|---|---|---|---|
| John dropped the milk. | 0.06 | 0.00 | 0.00 |
| John took the milk there. | 0.88 | 1.00 | 0.00 |
| Sandra went back to the bathroom. | 0.00 | 0.00 | 0.00 |
| John moved to the hallway. | 0.00 | 0.00 | 1.00 |
| Mary went back to the bedroom. | 0.00 | 0.00 | 0.00 |
| **Where is the milk?**　答案: **hallway**　预测: **hallway** | | | |

图　9-3

### 5. 输出模块

输出模块通常取决于手头的任务。在本例中，它被用于从一组候选项中检索最合适的答复。

为此，它首先通过与输入模块和问题模块相同的方式将每个候选对象转换为嵌入，然后从存储模块中获取每个候选项嵌入与上下文向量的点积。对于每个候选项，我们都会获得其与上下文向量的相似度或匹配分数。为了进行推理，对所有候选者的相似度值应用 softmax 函数以选择最为合适的那个：

```
        def _output_module(self, context_vector):
            with tf.variable_scope("OuptutModule"):
                candidates_emb =
    tf.nn.embedding_lookup(self.output_word_emb_matrix,
                                                        self._candidates_vec)
                candidates_emb_sum = tf.reduce_sum(candidates_emb, 1)
                return tf.matmul(context_vector,
    tf.transpose(candidates_emb_sum))
```

如果任务需要生成响应而非检索，则可以使用 RNN 以类似于机器翻译任务的方式逐个词元地生成答案。

### 6. 整合起来

我们可以编写一个 inference 方法，将各个模块整合到同一个管道中，用于读取输入和问题、获取上下文向量并生成输出：

```
def _inference(self, facts, questions):
    with tf.variable_scope("MemoryNetwork"):
        input_vectors = self._input_module(facts)
        question_vectors = self._question_module(questions)
        context_vectors = self._memory_module(question_vectors,
                                              input_vectors)
        output = self._output_module(context_vectors)
        return output
```



最后，我们定义 fit 和 predict 函数。这些函数使用记忆网络作为较大管道的一部分来进行训练和预测。我们使用 feed_dict 将数据传递到初始化代码所定义的操作中，该操作随后将运行_inference 函数：

```
def fit(self, facts, questions, answers):
    feed_dict = {self._facts: facts,
                 self._questions: questions,
                 self._answers: answers}
    loss, _ = self._session.run([self.loss_op, self.train_op],
                                feed_dict=feed_dict)
    return loss

def predict(self, facts, questions):
    feed_dict = {self._facts: facts, self._questions: questions}
    return self._session.run(self.predict_op, feed_dict=feed_dict)
```

## 9.3　拓展记忆网络以进行对话建模

我们将对话视为两个参与者（例如 A 和 B）之间基于话轮的会话，其中每轮对话都包含 A 的发声（utterance）和 B 的响应（response）。我们可以将产生每轮的响应视为 NLU 任务，通过查询之前的整个会话历史为传入的查询选择或生成适当的响应。

我们已经讨论了如何构建基于记忆网络的 QA 模型。该模型将一个问题和一些相关事实作为输入，并通过对事实进行推理来生成对于该问题的响应。为了有效地将对话建模为此类框架的一部分，我们将每次对话时的发声视为一个问题输入，而整个对话历史则是事实。记忆网络将基于此产生响应，如图 9-4 所示。

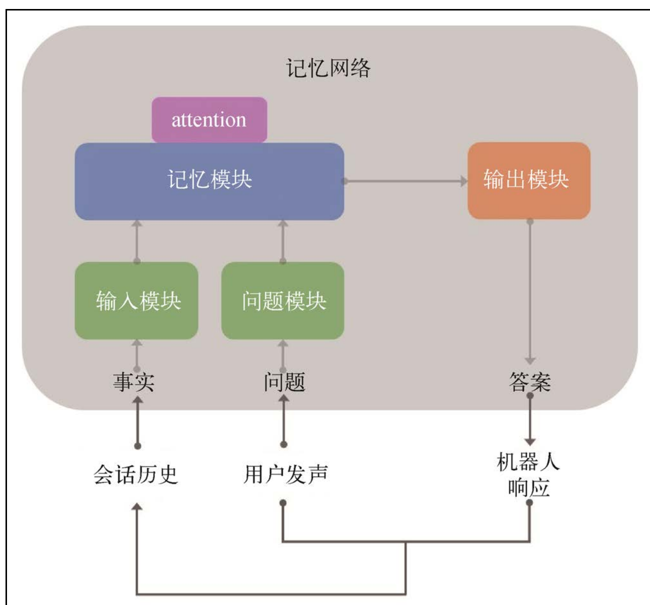



图　9-4

为了使对话在下一轮继续进行，先前的发声和响应对需要被添加到会话历史中去。然后该模型将被用于处理下一次发声并产生适当的响应，直至对话结束。

在深入研究记忆网络聊天机器人背后的代码之前，我们将介绍一些对话数据集，并特别讨论 Facebook AI 研究所用的 bAbI dialog 数据集。

### 9.3.1　对话数据集

对话任务通常分为两大类：开放式会话（也称为**聊天**）系统和面向目标系统。

**开放式会话系统**通常处理不受话题限制的对话，并使用来自 Twitter 会话、reddit 回复或类似论坛帖子的大规模语料进行训练。大多数开放式任务需要生成响应，因此大多数模型使用 seq2seq 框架，类似于机器翻译或文本摘要，并结合翻译指标（例如 BLEU 得分）和人工来评估。

除了语言建模和生成之外，构建这些神经会话模型所涉及的主要挑战是缺乏一致性，因为模型是在许多不同说话者之间的对话中进行训练，并且倾向于产生不明确的答案（例如“我不知道”）。

**面向目标对话系统**则旨在用于用户和机器人之间的特定交互，例如客户服务、餐厅预订、电影预订或其他礼仪服务。模型通过插空填充来预测对话状态或在每次对话框打开时选择最为合适的响应，我们可以通过这些能力对模型开展评估。

面向目标系统的主要挑战是将先验知识、会话历史记录和上下文结合起来，以实现所设定的目标。因此，最常见的架构包括上一节所说的用于开展会话的扩展 QA 模型。

### bAbI dialog 数据集

bAbI dialog 数据集（由 Bordes 等人于 2016 年引入）是最简单的面向目标对话数据集，旨在测试饭店预订领域中端对端训练的系统。如先前所述，对话任务对 bAbI 任务进行了补充，有助于文本理解。



关于创建和使用 bAbI dialog 数据集的完整信息可以在 Antoine Bordes、Y-Lan Boureau 和 Jason Weston 的论文“Learning End-to-End Goal-Oriented Dialog”中找到。

将领域设定在餐厅预定方面，合成生成的数据集将机器人与用户之间的对话分解为五个任务，以测试对话系统应具备的某些关键功能。给定包含餐厅及其属性（位置、美食类型等）的**知识库**（knowledge base，KB），对话的目的就是为用户预订餐厅。完整的对话分为多个阶段，每个阶段测试模型是否可以学习到某些能力，例如隐式对话状态跟踪、在对话中使用知识库事实，以及处理未出现在训练集对话中的新实体。

图 9-5 能使你对任务有一个更好的理解。

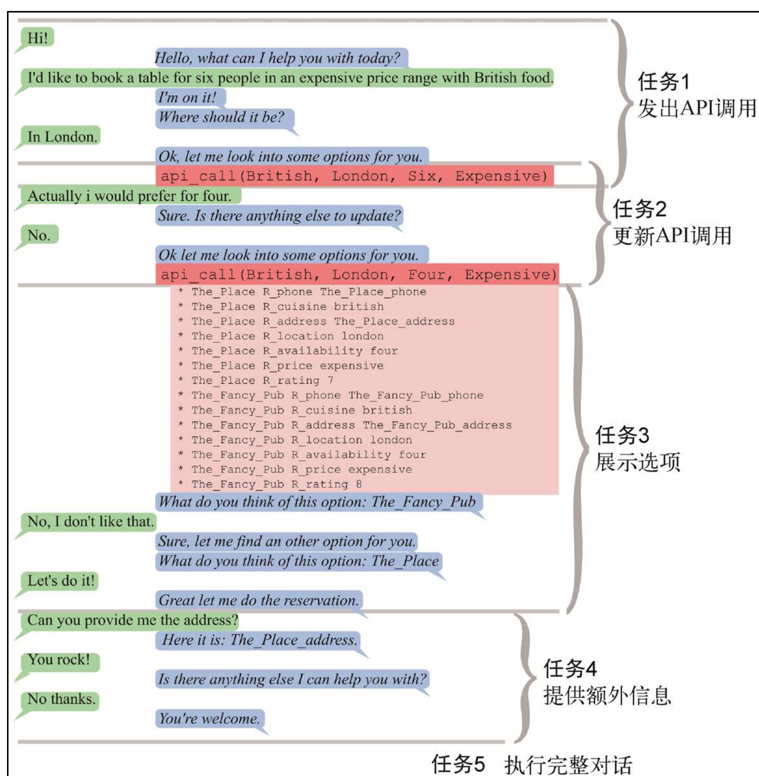



图　9-5

会话是由模拟器（以固定模板格式）基于包含所有餐厅及其属性的基本知识库生成的。每个餐厅都由美食类型（10 种选择，例如意大利、印度风味）、位置（10 种选择，例如伦敦、东京）、价格范围（便宜、适中或昂贵）、聚会规模（2 人、4 人、6 人或 8 人）和一个评分（从 1 到 8）来定义。每个餐厅也都有一个地址和电话号码。对知识库进行 API 调用将返回与所有满足以下四个参数的餐厅有关的事实列表：位置、美食类型、价格范围和聚会规模。除了用户和机器人的发声之外，每个任务中的对话还包括 API 调用和所产生的事实。在随机选择四个必填字段（位置、美食类型、价格范围和聚会规模）之后，使用自然语言模式生成会话。用户有 43 种模式，而机器人有 15 种模式（用户可以用四种不同的方式说话，而机器人的说话方式只有一种）。

尽管这些任务旨在于面向目标的环境中用作分析对话系统缺点的框架，但我们将专注于第五项任务：进行完整的会话。该任务将前四个任务的所有方面组合到完整的对话脚本中，并且可以训练一个简单的聊天机器人来预订餐厅。

- ***原始数据格式***

五个任务各自包含 1000 个对话，分别用于训练、验证和测试。每个任务的文件格式如下：

```
ID user_utterance [tab] bot_response
...
```

给定对话的 ID 从 1 开始并随着对话而递增。当文件中的 ID 被重新设置为 1 时，之后的句子将会作为新对话的开始。以下是原始对话数据的示例：

```
1 hi       hello what can i help you with today
2 can you make a restaurant reservation with french cuisine for four
  people in an expensive price range      i'm on it
3 &lt;SILENCE&gt;       where should it be
4 tokyo please   ok let me look into some options for you
5 &lt;SILENCE&gt;       api_call french tokyo four expensive
```

该模型必须学会预测用户发声后的机器人响应。响应可以是一个句子或一个 API 调用（以 api_call 开头）。

## 9.3.2 使用TensorFlow编写一个聊天机器人

在以下各节中，我们将通过管道使用 bAbI dialog 数据集进行训练并与记忆网络聊天机器人进行交互：我们将加载并处理数据，以使其与记忆网络框架兼容，然后围绕所述模型编写封装器，最后训练我们的聊天机器人。

### 1. 以 QA 格式导入对话数据集

如上一节所述，我们需要在每次对话时将逐行会话的对话数据转换为（事实，问题，答案）的元组格式。为此，需要编写一个方法，从原始对话语料库中读取行数据，并返回所需的元组以在记忆网络范式中进行训练。

由于我们将使用词向量作为模型的输入，首先需要定义一个 tokenize 方法，将句子转换为单词列表（除去特殊符号和常用单词）：



```
def tokenize(sent):
    stop_words = {"a", "an", "the"}
    sent = sent.lower()
    if sent == '&lt;silence&gt;':
        return [sent]
    # 将句子转变为词元
    result = [word.strip() for word in re.split('(\W+)?', sent)
              if word.strip() and word.strip() not in stop_words]
    # 清理
    if not result:
        result = ['&lt;silence&gt;']
    if result[-1]=='.' or result[-1]=='?' or result[-1]=='!':
        result = result[:-1]
    return result
```

然后，我们可以定义一个函数来从 bAbI dialog 数据集中读取原始数据文件并进行处理。我们逐行解析文件中的文本，并跟踪数据中所有潜在的（事实，问题，答案）元组。当我们从一行移

到下一行时，将不断更新对话框中的事实列表，并在遇到空白行时将其重置，还需注意不包含（发声，响应）对的行：

```
def parse_dialogs_per_response(lines, candidates_to_idx):
    data = []
    facts_temp = []
    utterance_temp = None
    response_temp = None
    # 逐行解析
    for line in lines:
        line = line.strip()
        if line:
            id, line = line.split(' ', 1)
            if '\t' in line: # 有发声和响应
                utterance_temp, response_temp = line.split('\t')
                # 将答案转换为整数索引
                answer = candidates_to_idx[response_temp]
                # 句子分词
                utterance_temp = tokenize(utterance_temp)
                response_temp = tokenize(response_temp)
                # 将(事实, 问题, 答案)元组添加到数据中
                data.append((facts_temp[:], utterance_temp[:], answer))
                # 增加发声/响应编码
                utterance_temp.append('$u')
                response_temp.append('$r')
                # 增加话轮计数时序编码
                utterance_temp.append('#' + id)
                response_temp.append('#' + id)
                # 更新事实
                facts_temp.append(utterance_temp)
                facts_temp.append(response_temp)
            else: # 有知识库事实
                response_temp = tokenize(line)
                response_temp.append('$r')
                response_temp.append('#' + id)
                facts_temp.append(response_temp)
        else: # 新对话
            facts_temp = []
    return data
```

需要注意一个重要而细微的差别：我们在数据里所有事实、问题和响应的标记化版本中添加了两个额外的符号（发声/响应编码和话轮计数编码）。这导致我们的模型将这些编码也视为单词并为其建立了词向量。话语/响应编码有助于模型区分用户和机器人说出的句子，而话轮计数编码则可以在模型中建立时序上的理解。

在这里，`candidates` 字典是候选答案到整数索引的映射。我们需要进行这样的转换，因为记忆网络将对候选项、字典整数条目执行 softmax 运算，然后将其指向所选的响应。我们可以直接从包含所有可能响应候选文件的文件中逐行构建 `candidates` 字典，以及响应候选本身的标记化版本，如以下代码所示：

```
candidates = []
candidates_to_idx = {}
with open('dialog-babi/dialog-babi-candidates.txt') as f:
    for i, line in enumerate(f):
        candidates_to_idx[line.strip().split(' ', 1)[1]] = i
        line = tokenize(line.strip())[1:]
        candidates.append(line)
```

接下来，使用刚定义的解析方法可以通过 candidates 字典来载入 QA 格式的训练、验证和测试对话。

```
train_data = []
with open('dialog-babi/dialog-babi-task5-full-dialogs-trn.txt') as f:
    train_data = parse_dialogs_per_response(f.readlines(),
candidates_to_idx)

test_data = []
with open('dialog-babi/dialog-babi-task5-full-dialogs-tst.txt') as f:
    test_data = parse_dialogs_per_response(f.readlines(),
candidates_to_idx)

val_data = []
with open('dialog-babi/dialog-babi-task5-full-dialogs-dev.txt') as f:
    val_data = parse_dialogs_per_response(f.readlines(), candidates_to_idx)
```

### 2. 向量化数据

数据预处理的最终阶段是向量化或量化对话和候选项。这需要将每个单词或标记转换为整数值，即将单词的任何序列转换为与每个单词相对应的整数序列。

我们首先编写一种向量化候选文本的方法，还必须牢记每个向量化候选对象的固定单词长度（sentence_size）。因此，我们需要用 0 来填充长度小于所需大小的候选向量：

```
def vectorize_candidates(candidates, word_idx, sentence_size):
    # 确定最终向量的形状
    shape = (len(candidates), sentence_size)
    candidates_vector = []
    for i, candidate in enumerate(candidates):
        # 确定零填充
        zero_padding = max(0, sentence_size - len(candidate))
        # 附加到最终向量末尾
        candidates_vector.append(
            [word_idx[w] if w in word_idx else 0 for w in candidate]
            + [0] * zero_padding)
    # 以 TensorFlow 常量返回
    return tf.constant(candidates_vector, shape=shape)
```

接下来，我们将以类似的方式编写一种方法以向量化对话数据。我们需要关注的另一个重要方面是确保将每个数据样本的事实向量与空记忆向量（sentence_size 为 0 的向量）填充至固定的记忆大小：

```
def vectorize_data(data, word_idx, sentence_size, batch_size,
max_memory_size):
    facts_vector = []
    questions_vector = []
    answers_vector = []
    # 对数据根据事实量降序排序
    data.sort(key=lambda x: len(x[0]), reverse=True)
    for i, (fact, question, answer) in enumerate(data):
        # 找到存储大小
        if i % batch_size == 0:
            memory_size = max(1, min(max_memory_size, len(fact)))
        # 构建事实向量
        fact_vector = []
        for i, sentence in enumerate(fact, 1):
            fact_padding = max(0, sentence_size - len(sentence))
            fact_vector.append(
                [word_idx[w] if w in word_idx else 0 for w in sentence]
                + [0] * fact_padding)
        # 保存适合记忆的最新句子
        fact_vector = fact_vector[::-1][:memory_size][::-1]
        # 填充到 memory_size
        memory_padding = max(0, memory_size - len(fact_vector))
        for _ in range(memory_padding):
            fact_vector.append([0] * sentence_size)
        # 建立问题向量
        question_padding = max(0, sentence_size - len(question))
        question_vector = [word_idx[w] if w in word_idx else 0
                           for w in question] \
                           + [0] * question_padding
        # 附加到最终向量末尾
        facts_vector.append(np.array(fact_vector))
        questions_vector.append(np.array(question_vector))
        # 答案已经是对应于一个候选项的整数了
        answers_vector.append(np.array(answer))
    return facts_vector, questions_vector, answers_vector
```

要强调的是需要事先对这些维度有所了解，因为我们会将这些向量发送到 TensorFlow 模型，而该模型需要知道输入的大小才能构建模型图。

### 3. 在聊天机器人类中封装记忆网络模型

我们将把数据提供给通用的聊天机器人类，并调用其中的向量化方法。我们将使用它作为先前定义的记忆网络模型的封装器。从理论上讲，该模型可以被替换成其他任何基于 QA 的模型。

- **类构造器**

使用 `class` 构造函数，我们可以加载数据和候选项，构建词汇表，然后初始化 TensorFlow 会话和记忆网络对象：

```
class ChatBotWrapper(object):
    def __init__(self, train_data, test_data, val_data,
```

```
                  candidates, candidates_to_idx,
                  memory_size, batch_size, learning_rate,
                  evaluation_interval, hops,
                  epochs, embedding_size):
         self.memory_size = memory_size
         self.batch_size = batch_size
         self.evaluation_interval = evaluation_interval
         self.epochs = epochs

         self.candidates = candidates
         self.candidates_to_idx = candidates_to_idx
         self.candidates_size = len(candidates)
         self.idx_to_candidates = dict((self.candidates_to_idx[key], key)
                                       for key in self.candidates_to_idx)
         # 初始化数据并建立词汇表
         self.train_data = train_data
         self.test_data = test_data
         self.val_data = val_data
         self.build_vocab(train_data + test_data + val_data, candidates)
         # 向量化候选项
         self.candidates_vec = vectorize_candidates(
             candidates, self.word_idx, self.candidate_sentence_size)
         # 初始化优化器
         optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
         # 初始化 TensorFlow 会话和记忆网络模型
         self.sess = tf.Session()
         self.model = MemoryNetwork(
                         self.sentence_size, self.vocab_size,
                         self.candidates_size, self.candidates_vec,
                         embedding_size, hops,
                         optimizer=optimizer, session=self.sess)
```

- **为词嵌入查找建立词汇表**

我们希望为 facts、candidates 和 questions 中的每个单词创建词嵌入。因此，需要读取数据和候选项以计算要创建词嵌入的单词数以及最大的句子长度。此信息将被传递到内存网络模型以初始化嵌入矩阵和输入占位符：

```
     def build_vocab(self, data, candidates):
         # 从所有数据和候选词中建立单词词汇表
         vocab = reduce(lambda x1, x2: x1 | x2,
             (set(list(chain.from_iterable(facts)) + questions)
                 for facts, questions, answers in data))
         vocab |= reduce(lambda x1, x2: x1 | x2,
             (set(candidate) for candidate in candidates))
         vocab = sorted(vocab)
         # 为每个单词分配整数索引
         self.word_idx = dict((word, idx + 1) for idx, word in
enumerate(vocab))
         # 计算有多少个不同数据大小
         max_facts_size = max(map(len, (facts for facts, _, _ in data)))
         self.sentence_size = max(
```

```
            map(len, chain.from_iterable(facts for facts, _, _ in data)))
        self.candidate_sentence_size = max(map(len, candidates))
        question_size = max(map(len, (questions for _, questions, _ in
data)))
        self.memory_size = min(self.memory_size, max_facts_size)
        self.vocab_size = len(self.word_idx) + 1 # +1 for null word
        self.sentence_size = max(question_size, self.sentence_size)
```

- **训练聊天机器人模型**

我们可以将向量化的训练数据（由在上一节中定义的向量化方法所得到）传递给聊天机器人，并调用记忆网络的 `fit` 方法来训练小批次训练数据，同时以固定的间隔在验证集上评估模型的性能：

```
def predict_for_batch(self, facts, questions):
    preds = []
    # 在小批次上迭代
    for start in range(0, len(facts), self.batch_size):
        end = start + self.batch_size
        facts_batch = facts[start:end]
        questions_batch = questions[start:end]
        # 对每个批次进行预测
        pred = self.model.predict(facts_batch, questions_batch)
        preds += list(pred)
    return preds

def train(self):
    # 向量化训练数据和验证数据
    train_facts, train_questions, train_answers = vectorize_data(
        self.train_data, self.word_idx, self.sentence_size,
        self.batch_size, self.memory_size)
    val_facts, val_questions, val_answers = vectorize_data(
        self.val_data, self.word_idx, self.sentence_size,
        self.batch_size, self.memory_size)
    # 将训练数据分批
    batches = zip(range(0, len(train_facts) - self.batch_size,
                        self.batch_size),
                  range(self.batch_size, len(train_facts),
                        self.batch_size))
    batches = [(start, end) for start, end in batches]
    # 开始训练循环
    for epoch in range(1, self.epochs + 1):
        np.random.shuffle(batches)
        total_cost = 0.0
        for start, end in batches:
            facts = train_facts[start:end]
            questions = train_questions[start:end]
            answers = train_answers[start:end]
            # 在批数据上开展训练
            batch_cost = self.model.fit(facts, questions, answers)
            total_cost += batch_cost
        if epoch % self.evaluation_interval == 0:
```

```
            # 在训练集和验证集上计算准确率
            train_preds = self.predict_for_batch(
                train_facts, train_questions)
            val_preds = self.predict_for_batch(
                val_facts, val_questions)
            train_acc = metrics.accuracy_score(
                train_preds, train_answers)
            val_acc = metrics.accuracy_score(
                val_preds, val_answers)
            print("Epoch: ", epoch)
            print("Total Cost: ", total_cost)
            print("Training Accuracy: ", train_acc)
            print("Validation Accuracy: ", val_acc)
            print("---")
```

- **在测试集上评估聊天机器人**

然后，我们可以编写一个方法来预测测试数据集中每个对话的响应并获得准确率得分：

```
def test(self):
    # 在测试集上计算准确率
    test_facts, test_questions, test_answers = vectorize_data(
        self.test_data, self.word_idx, self.sentence_size,
        self.batch_size, self.memory_size)
    test_preds = self.predict_for_batch(test_facts, test_questions)
    test_acc = metrics.accuracy_score(test_preds, test_answers)
    print("Testing Accuracy: ", test_acc)
```

- **与聊天机器人交互**

最后，我们可以按照前面各节中所描述的框架与聊天机器人进行交互。在每个用户发声之后，我们要求记忆网络根据历史会话记录和用户发声来预测响应，随后将发声和响应添加到历史会话记录中，之后用户可以再次发声：

```
def interactive_mode(self):
    facts = []
    utterance = None
    response = None
    turn_count = 1
    while True:
        line = input("==&gt; ").strip().lower()
        if line == "exit":
            break
        if line == "restart":
            facts = []
            turn_count = 1
            print("Restarting dialog...\n")
            continue
        utterance = tokenize(line)
        data = [(facts, utterance, -1)]
        # 将数据向量化并做出预测
        f, q, a = vectorize_data(data, self.word_idx,
```

```
            self.sentence_size, self.batch_size, self.memory_size)
        preds = self.model.predict(f, q)
        response = self.idx_to_candidates[preds[0]]
        # 打印预测响应
        print(response)
        response = tokenize(response)
        # 加入话轮计数时序编码
        utterance.append("$u")
        response.append("$r")
        # 加入发声/响应编码
        utterance.append("#" + str(turn_count))
        response.append("#" + str(turn_count))
        # 更新事实存储
        facts.append(utterance)
        facts.append(response)
        turn_count += 1
```

- **整合起来**

为了运行刚刚编写的代码，我们将定义模型的超参数并实例化 chatbot 模型。随后，我们将开始对模型进行 200 个周期的训练并每隔 10 个周期在验证集上评估其性能。经过训练后，我们可以在测试数据上对模型进行测试，代码如下所示：

```
chatbot = ChatBotWrapper(train_data, test_data, val_data,
                         candidates, candidates_to_idx,
                         memory_size=50,
                         batch_size=32,
                         learning_rate=0.001,
                         evaluation_interval=10,
                         hops=3,
                         epochs=100,
                         embedding_size=50)
chatbot.train()
chatbot.test()
```

以下是输出：

```
Epoch: 10
Total Cost: 17703.9733608
Training Accuracy: 0.756870229008
Validation Accuracy: 0.729912770223
------------------------------------------------
Epoch: 20
Total Cost: 7439.67566451
Training Accuracy: 0.903217011996
Validation Accuracy: 0.857127377147
------------------------------------------------
Epoch: 30
Total Cost: 3179.78263753
Training Accuracy: 0.982769901854
Validation Accuracy: 0.939372595763
.
.
.
```

```
------------------------------------------------
Epoch: 80
Total Cost: 1949.99280906
Training Accuracy: 0.980861504907
Validation Accuracy: 0.937747196186
------------------------------------------------
Epoch: 90
Total Cost: 500.894205613
Training Accuracy: 0.995637949836
Validation Accuracy: 0.95400119196
------------------------------------------------
Epoch: 100
Total Cost: 912.067172846
Training Accuracy: 0.995092693566
Validation Accuracy: 0.954813891748
-------------------------------------------------
Testing Accuracy:  0.958093271008
```

在训练 chatbot 时，可以根据其在验证数据上的表现评估其性能。应该可以看到，它的损失在不断减少且准确率有所提高。尽管采用更严格的正则化方案（例如梯度裁剪、L2 范数正则化或随机失活）会得到更好的结果，但在训练结束时，我们仍可以得到一个在测试集上具有良好表现的模型。使用 TensorFlow 添加这些正则方案相当简单，你可以自行练习。

- **交互对话示例**

我们也可以在交互模式下运行训练好的聊天机器人以进行实时对话：

```
chatbot.interactive_mode()
```

每当出现==>符号提示时，都可以键入文本来与聊天机器人进行交互：

```
==&gt; good morning
hello what can i help you with today
==&gt; i'd like to book a table for eight
i'm on it
==&gt;
any preference on a type of cuisine
==&gt; with italian cuisine
where should it be
==&gt; in bombay
which price range are looking for
==&gt; i am looking for a moderate restaurant
ok let me look into some options for you
==&gt;
api_call italian bombay eight moderate
==&gt; instead could it be in a cheap price range
sure is there anything else to update
==&gt; actually i would prefer in london
sure is there anything else to update
==&gt; no
ok let me look into some options for you
==&gt;
```

```
api_call italian london eight cheap
==&gt;
.
.
.
```

当我们尝试各种类型的交互时，可以看到这个简单的记忆网络已经学会了对大多数问题做出适当的回答，但是在处理专有名词或命名实体（例如餐馆、存储在知识库中的相关事实）时可能会稍显笨拙。与这样的实体打交道本身也可以被认为是一条研究线，而且需要结合 NLP 研究的许多子领域来构建用于实际部署的聊天机器人。

### 9.3.3　记忆网络相关文献

对于求知欲较强的读者，我们提供了一份论文列表，这些论文介绍了与记忆网络相关或受其启发的新思想和新架构，如表 9-2 所示。

表　9-2

| 标　　题 | 描　　述 |
| --- | --- |
| “Dynamic Memory Networks (DMNs) and Dynamic Coattention Networks (DCNs)” | DMN 是由 Salesforce Research 推出的（与 Facebook 的记忆网络几乎同时），它使用更为复杂的 RNN 来进行构建表示和迭代情境记忆。DCN 是 Salesforce 对基于 attention 的推理模型的最新迭代，具有全新的 coattention 机制 |
| “Neural Turing Machines (NTMs) and Differentiable Neural Computer (DNC)” | DeepMind 的 NTM 和 DNC 设定了更为积极的目标：使神经网络可以读取和写入外部存储并执行计算机可以执行的任何算法 |
| “Seq2seq Memory Network” | Microsoft Research 引入了用于生成对话框的 seq2seq 模型，并为其增加了与记忆网络非常相似的记忆模块 |
| “Recurrent Entity Networks” | Facebook 基于 attention 模型的最新迭代，与内存网络相反，它可以即时建立内存并对其进行推理 |

## 9.4　小结

本章，我们快速介绍了 QA 来了解自然语言理解问题，并学习了如何为任何 QA 任务构建通用的记忆网络模型。然后，我们研究了将会话建模作为 QA 任务的问题，并扩展了记忆网络以训练面向目标的聊天机器人。

我们构建了一个基于检索的简单聊天机器人以帮助用户根据喜好来预订餐厅。你可以进一步探索可能在某些方面更为复杂的 attention 机制，更强大的句子表示编码器，以及使用生成模型代替检索方法。

在下一章中，我们将介绍使用编码器–解码器模型进行语言翻译，并介绍更为复杂的、用于序列比对的 attention 机制。

第 10 章

# 使用基于 attention 的模型进行机器翻译

机器翻译系统将文本从一种语言转换为另一种语言，一个这样的系统就是 Google Translate 服务。本章将研究这些系统的架构及构建方法。虽然本章的重点在**神经机器翻译**上，但还是将简要介绍用于应对机器翻译挑战的传统方法。我们主要关注以下主题：

- 机器翻译概述
- 神经机器翻译概述
- 基于 attention 机制开发并训练一个神经机器翻译模型

## 10.1 机器翻译概述

目前有多种类型的机器翻译方法被广泛使用，但为简洁起见，我们将只研究两种主要方法：一是**统计机器翻译**（statistical machine translation，SMT），二是**神经机器翻译**（neural machine translation，NMT），后者也是本章的主题。我们将简要介绍以上两种方法。

### 10.1.1 统计机器翻译

统计机器翻译（SMT）将翻译模型与目标语言模型相结合，将句子从源文本使用的语言转换为目标语言，图 10-1 对此进行了说明。翻译模型将单词和短语从源语言映射到目标语言，而语言模型则捕获有关单词在目标语言中遵循特定顺序可能性的统计信息。因此，SMT 试图最大化选定目标语句（即源句子对应翻译）的可能性。这些统计模型来自于大量源语言到目标语言的翻译语料库：

在 SMT 之前，机器翻译要靠专家定义语言和句法规则以使翻译工作顺利进行。SMT 是自动机器翻译的一大进步，其规则是从大量双语数据中统计得出的。但 SMT 的主要困难之一在于，它只在翻译类似于训练语料库或对应领域的文本时才能表现良好，而对于来自不同领域的新输入文本，SMT 可能无法很好地翻译。SMT 的另一个缺点是需要大量的双语训练数据，而稀有的语

言对可能很难获取。同时，SMT 针对每种源语言到目标语言的翻译都需要独立、专门的管道。

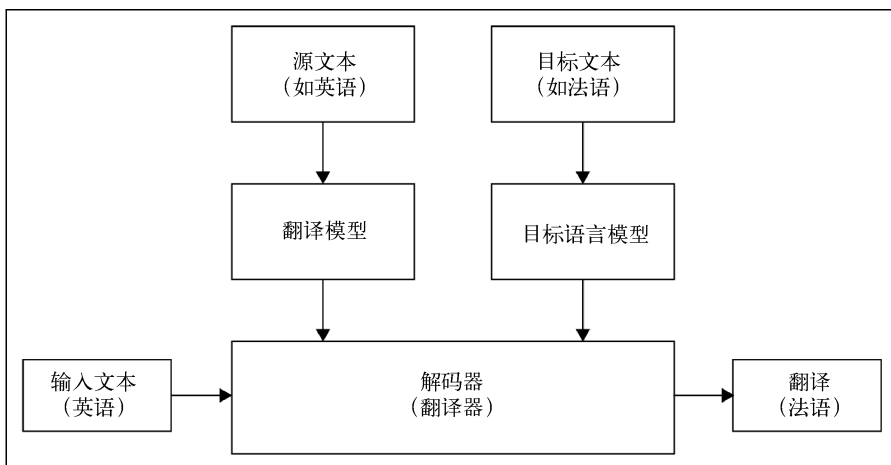



图　10-1

### 使用 NLTK SMT 模型实现英语到法语的翻译

现在，我们将研究一个使用 NLTK 进行统计机器翻译的示例。我们使用 TED 演讲的翻译作为训练数据集和测试数据集（这些数据包含一些由法语译成英语的 TED 演讲）。本书代码库的 Chapter10 目录下提供了本例的完整代码和数据。我们将使用 IBM 词法对齐模型（一种简单的统计转换模型）来获取源语言和目标语言之间的对齐对的集合，并计算其关联或对齐的概率。我们还将使用基本的 IBM Model 1，它会对源语句和目标语句进行一对一的对齐。因此，该模型能为每个源单词精确地生成一个目标单词，而无须考虑将源单词重新排序或翻译多词少词的情况。

`nltk.translate` 包提供了 IBM 对齐模型的实现。我们首先将其导入并定义一个函数以读取英语和相应的法语翻译数据：

```
from nltk.translate.ibm1 import IBMModel1
from nltk.translate.api import AlignedSent
import dill as pickle
import randomdef

read_sents(filename):
    sents = []
    c=0
    with open(filename,'r') as fi:
        for li in fi:
            sents.append(li.split())
    return sents
```

`AlignedSent` 类将在训练期间提供法语–英语对齐数据。`read_sents()`从输入文件中读取每一行，并将其转换为每个句子的词元列表。现在，我们将创建对齐化数据并训练模型：

```
max_count=5000
eng_sents_all = read_sents('data/train_en_lines.txt')
fr_sents_all = read_sents('data/train_fr_lines.txt')
eng_sents = eng_sents_all[:max_count]
fr_sents = fr_sents_all[:max_count]
print("Size of english sentences: ", len(eng_sents))
print("Size of french sentences: ", len(fr_sents))
aligned_text = []
for i in range(len(eng_sents)):
    al_sent = AlignedSent(fr_sents[i],eng_sents[i])
    aligned_text.append(al_sent)
print("Training smt model")
ibm_model = IBMModel1(aligned_text,5)
print("Training complete")
```

我们使用大约 5000 个句子（`max_count`）作为训练数据以加快收敛速度，但你也可以对该值进行修改以在完整数据上进行训练。然后，使用 `AlignedSent` 创建法语–英语句子对的列表来进行模型训练。在训练后，我们将研究模型在翻译任务中的表现：

```
n_random = random.randint(0,max_count)
fr_sent = fr_sents_all[n_random]
eng_sent_actual_tr = eng_sents_all[n_random]
tr_sent = []
for w in fr_sent:
    probs = ibm_model.translation_table[w]
    if(len(probs)==0):
        continue
    sorted_words = sorted([(k,v) for k, v in probs.items()],key=lambda x:
x[1], reverse=True)
    top_word = sorted_words[1][0]
    if top_word is not None:
        tr_sent.append(top_word)
print("French sentence: ", " ".join(fr_sent))
print("Translated Eng sentence: ", " ".join(tr_sent))
print("Original translation: ", " ".join(eng_sent_actual_tr))
```

可以从法语句子列表中随机选择一个句子，然后使用 `translation_table` 表查找相应的英语单词。该表存储了给定的法语单词和相应的英语单词之间对齐的可能性。我们将使用这些对齐概率来选择英语单词，其中对齐概率最高的单词更有可能是给定法语单词的翻译。我们将对原始句子中的所有法语单词进行查找，以在 `tr_sent` 中获得相应的英语短语。最后，打印法语句子、SMT 翻译的句子以及正确的翻译：

```
French sentence:  On appelle ça l'accessibilité financière.
Translated Eng sentence:  suggests affordability. works. called called
Original translation:  And it's called affordability.
```

可以看到，SMT 可以正确翻译某些单词（例如 affordability），但与原始句子相比，翻译并无实际意义。这一点可以通过在整个数据集上训练并增加迭代次数来改善。但是应该注意到，此处我们使用了一个不考虑目标语言中单词顺序的简单模型。更复杂的 IBM 模型（模型序号分别为 3、

4 和 5）可以捕获单词顺序和扩张能力（源语言单词并不总与目标语言单词具有一对一映射）。IBM 模型 5 使用**隐马尔可夫模型**（Hidden Markov Model，HMM）和对齐以提供更好的翻译。

## 10.1.2　神经机器翻译

**神经机器翻译**（NMT）使用神经网络来学习将文本从源语言翻译成目标语言。与 SMT 不同，NMT 的一个主要优点是只需要一种模型就可以端到端地实现语言转换。更重要的是，NMT 适用于源文本的整个片段，而非 SMT 中的文本块或短语。这一点是通过词嵌入学习上下文来实现的。因此，NMT 会在保留原始文本上下文的同时执行翻译。现在，我们将介绍 NMT 使用的一些常见的深度学习架构。

### 1. 编码器–解码器网络

最常见的架构类型是编码器–解码器网络，与我们在第 8 章中使用的相似。实际上，这两种模型架构并没有太大不同。该架构首先将源文本短语输入到编码器中，编码器会将其转换为代表短语含义的 thought vector，然后将这种密集表示形式连同训练期间的目标语言原始翻译一同馈入解码器以充当解码器的预处理，而解码器会根据训练时提供的原始翻译学习相应的翻译。

如 Luong 等人在论文“Effective Approaches to Attention-Based Neural Machine Translation”中所述，编码器–解码器网络如图 10-2 所示。图中的翻译是从源语言文本（英语）到目标语言文本（法语）的转换。

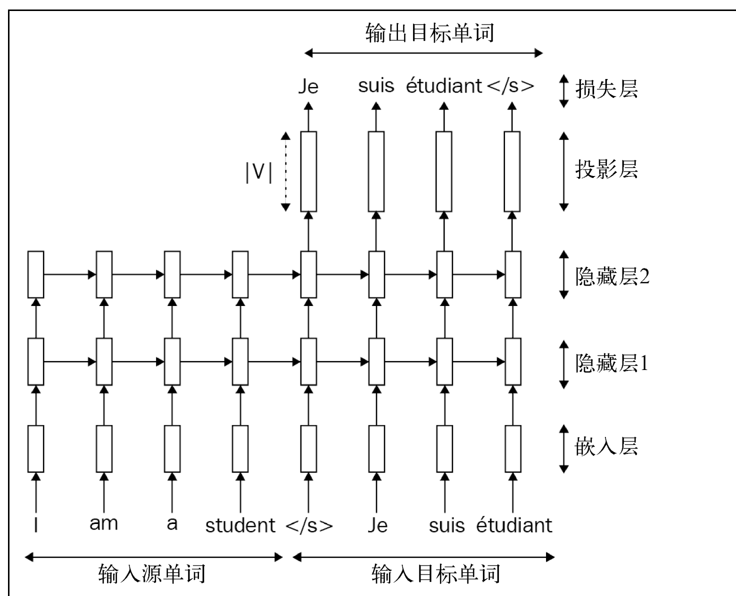



图　10-2

由于输入和输出的顺序性质，编码器和解码器的常见选择都是 RNN。我们通常使用 LSTM 或 GRU 来捕获源文本和目标文本中的长期关联。编码器 RNN 逐字读取源语言短语并生成最终状态。此最终状态以压缩向量表示的形式封装了短语的含义。该向量表示形式会被作为解码器的初始状态与目标短语一起馈入解码器，解码器则将如此学习翻译。

在推理期间，编码器使用从源语言短语获得的解码器最终状态压缩表示，以目标语言逐词输出短语。在图 10-2 中，嵌入层将单词转换为密集表示，然后由解码器转换为目标词汇表中所有单词的投影，并根据 softmax 概率从投影中选择最终用于翻译的单词。

### 2. 使用 attention 的编码器–解码器架构

上面描述的编码器–解码器架构有一个主要缺点：最终的编码器状态为固定长度，因此可能导致信息丢失。虽然这对于较短的语句来说可能不是问题，但对于较长的源语言输入，编码器可能无法捕获长期依存关系，从而导致解码器无法输出良好的翻译。为了克服这个问题，Bahdanau 等人在论文“Neural Machine Translation by Jointly Learning to Align and Translate”中引入了 attention 机制。图 10-3 是从其论文中摘录的架构图示。

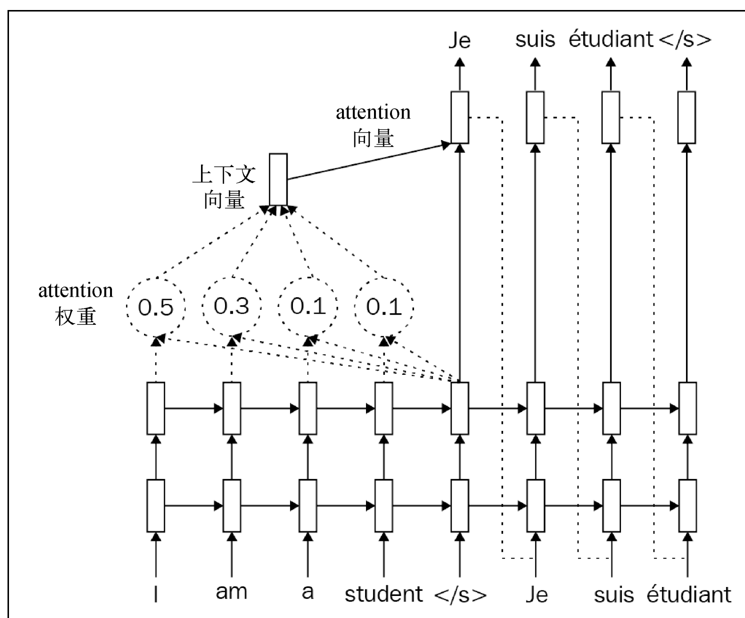



图 10-3

attention 机制的主要思想是在学习翻译时集中注意力于输入源文本的重要部分。实际上，attention 机制通过在训练过程中获得的权重，在输入源文本和目标文本之间建立快捷连接。这些连接增强了解码器翻译较长输入短语的能力，从而使翻译更为准确。

### 3. 使用 attention 进行从法语到英语的神经机器翻译

这里将使用与 SMT 中相同的数据集，并将基于 attention 机制来构建网络。你还将发现该网络类似于第 8 章中所描述的架构。该示例的完整 Jupyter Notebook 可在本书代码库中找到（Chapter10/02_example.ipynb）。

- **数据准备**

首先，分别读入法语源文本和英语目标文本：

```
frdata=[]
endata=[]
with open('data/train_fr_lines.txt') as frfile:
    for li in frfile:
        frdata.append(li)
with open('data/train_en_lines.txt') as enfile:
    for li in enfile:
        endata.append(li)
mtdata = pd.DataFrame({'FR':frdata,'EN':endata})
mtdata['FR_len'] = mtdata['FR'].apply(lambda x: len(x.split(' ')))
mtdata['EN_len'] = mtdata['EN'].apply(lambda x: len(x.split(' ')))
print(mtdata['FR'].head(2).values)
print(mtdata['EN'].head(2).values)

Output:

['Voici Bill Lange. Je suis Dave Gallo.\n'
 'Nous allons vous raconter quelques histoires de la mer en vidéo.\n']
["This is Bill Lange. I'm Dave Gallo.\n"
 "And we're going to tell you some stories from the sea here in video.\n"]
```

因为我们将使用预训练好的嵌入向量，所以将其载入以创建一个由单词到嵌入的字典。我们将使用该字典来准备用于训练的输入文本数据：

```
def build_word_vector_matrix(vector_file):
    embedding_index = {}
    with codecs.open(vector_file, 'r', 'utf-8') as f:
        for i, line in enumerate(f):
            sr = line.split()
    word = sr[0]
    embedding = np.asarray(sr[1:], dtype='float32')
    embedding_index[word] = embedding
    return embedding_index
embeddings_index = build_word_vector_matrix('glove.6B.50d.txt')
```

由于编码器和解码器的输入是单词标识符，我们将同时创建单词到 ID 和 ID 到单词的映射以在训练和推理中使用。在训练期间使用单词到 ID 的映射，而 ID 到单词的映射将用于在推理过程中得到翻译文本：

```
def build_word2id_mapping(word_counts_dict):
    word2int = {}
    count_threshold = 20
    value = 0
    for word, count in word_counts_dict.items():
        if count &gt;= count_threshold or word in embeddings_index:
            word2int[word] = value
            value += 1
    special_codes = [TOKEN_UNK,TOKEN_PAD,TOKEN_EOS,TOKEN_GO]
    for code in special_codes:
        word2int[code] = len(word2int)
    int2word = {}
    for word, value in word2int.items():
        int2word[value] = word
    return word2int,int2word
```

这会将输入单词和特殊标记 TOKEN_UNK、TOKEN_PAD、TOKEN_EOS 和 TOKEN_GO 转换为相应的数字标识符。这些特殊标记分别被定义为字符串 UNK、PAD、EOS 和 GO。我们将同时对于英法文本应用 build_word2id_mapping()和 build_embeddings()函数。请注意，仅使用出现频率大于 count_threshold（在先前中设置为 20）的单词：

```
fr_word2int,fr_int2word = build_word2id_mapping(word_counts_dict_fr)
en_word2int,en_int2word = build_word2id_mapping(word_counts_dict_en)
fr_embeddings_matrix = build_embeddings(fr_word2int)
en_embeddings_matrix = build_embeddings(en_word2int)
print("Length of french word embeddings: ", len(fr_embeddings_matrix))
print("Length of english word embeddings: ", len(en_embeddings_matrix))

Output:

Length of french word embeddings: 19708
Length of english word embeddings: 39614
```

接下来定义函数以将源短语和目标短语转换为数字标识符：

```
def convert_sentence_to_ids(text, word2int, eos=False):
    wordints = []
    word_count = 0
    for sentence in text:
        sentence2ints = []
        for word in sentence.split():
            word_count += 1
            if word in word2int:
                sentence2ints.append(word2int[word])
            else:
                sentence2ints.append(word2int[TOKEN_UNK])
    if eos:
        sentence2ints.append(word2int[TOKEN_EOS])
    wordints.append(sentence2ints)
return wordints, word_count
```

正如先前所做的那样，我们对源文本和输入文本应用 convert_sentence_to_ids()函数：

```
id_fr, word_count_fr = convert_sentence_to_ids(mtdata_fr, fr_word2int)
id_en, word_count_en = convert_sentence_to_ids(mtdata_en, en_word2int,
eos=True)
```

因为句子/短语中含有许多对训练无益的未知单词和词元，所以我们将其从集合中移除：

```
en_filtered = []
fr_filtered = []
max_en_length = int(mtdata.EN_len.max())
max_fr_length = int(mtdata.FR_len.max())
min_length = 4
unknown_token_en_limit = 10
unknown_token_fr_limit = 10
for count,text in enumerate(id_en):
    unknown_token_en = unknown_tokens(id_en[count],en_word2int)
    unknown_token_fr = unknown_tokens(id_fr[count],fr_word2int)
    en_len = len(id_en[count])
    fr_len = len(id_fr[count])
    if( (unknown_token_en&gt;unknown_token_en_limit) or
(unknown_token_fr&gt;unknown_token_fr_limit) or
        (en_len&lt;min_length) or (fr_len&lt;min_length) ):
        continue
    fr_filtered.append(id_fr[count])
    en_filtered.append(id_en[count])
print("Length of filtered french/english sentences: ", len(fr_filtered),
len(en_filtered) )

Output:

Length of filtered french/english sentences: 200404 200404
```

注意,我们移除了包含超过`unknown_token_en_limit`或`unknown_token_fr_limit`( 在代码中均设置为`10`) 个未知词元的句子。同样，还将长度小于 4 个单词的句子移除了。

- **编码器网络**

现在，我们将对概述部分描述的原始架构稍加修改来构建编码器网络。使用双向 RNN 代替单向 RNN 从而捕获输入中的前向和后向依赖关系：

```
def get_rnn_cell(rnn_cell_size,dropout_prob):
    rnn_c = GRUCell(rnn_cell_size)
    rnn_c = DropoutWrapper(rnn_c, input_keep_prob = dropout_prob)
    return rnn_c

def encoding_layer(rnn_cell_size, sequence_len, n_layers, rnn_inputs,
dropout_prob):
    for l in range(n_layers):
        with tf.variable_scope('encoding_l_{}'.format(l)):
            rnn_fw = get_rnn_cell(rnn_cell_size,dropout_prob)
            rnn_bw = get_rnn_cell(rnn_cell_size,dropout_prob)
    encoding_output, encoding_state =
tf.nn.bidirectional_dynamic_rnn(rnn_fw, rnn_bw, rnn_inputs,
```

```
                                        sequence_len,dtype=tf.float32)
    encoding_output = tf.concat(encoding_output,2)
    return encoding_output, encoding_state
```

我们使用 GRU 作为 RNN 的循环单元，并将输入语句的嵌入馈送到编码器 rnn_inputs。encoding_layer 函数的其他参数有单元大小、序列长度和 dropout 概率。稍后我们将序列长度设置为法语文本的最大长度。



请注意，我们已将前向 RNN 和后向 RNN 的编码输出级联在了一起，还使用了 DropoutWrapper 来将 dropout 合并到编码层中。

- **解码器网络**

解码器网络也由 GRU 单元创建。decoding_layer 函数将编码器的输出和英语文本的词嵌入作为输入，输出投影向量的大小等于英语文本的词汇量：

```
def decoding_layer(decoding_embed_inp, embeddings, encoding_op,
encoding_st, v_size, fr_len,             en_len,max_en_len, rnn_cell_size,
word2int, dropout_prob, batch_size, n_layers):
    for l in range(n_layers):
        with tf.variable_scope('dec_rnn_layer_{}'.format(l)):
            gru = tf.contrib.rnn.GRUCell(rnn_len)
            decoding_cell =
tf.contrib.rnn.DropoutWrapper(gru,input_keep_prob = dropout_prob)
    out_l = Dense(v_size, kernel_initializer =
tf.truncated_normal_initializer(mean = 0.0,
            stddev=0.1))
    attention = BahdanauAttention(rnn_cell_size, encoding_op,fr_len,
normalize=False,
name='BahdanauAttention')
    decoding_cell = AttentionWrapper(decoding_cell,attention,rnn_len)
    attention_zero_state = decoding_cell.zero_state(batch_size , tf.float32
)
    attention_zero_state = attention_zero_state.clone(cell_state =
encoding_st[0])
    with tf.variable_scope("decoding_layer"):
        logits_tr = training_decoding_layer(decoding_embed_inp, en_len,
decoding_cell,
attention_zero_state,out_l,v_size, max_en_len)
    with tf.variable_scope("decoding_layer", reuse=True):
        logits_inf = inference_decoding_layer(embeddings,
word2int[TOKEN_GO],word2int[TOKEN_EOS],
                        decoding_cell, attention_zero_state,
out_l,max_en_len,batch_size)
    return logits_tr, logits_inf
```

我们还使用 DropoutWrapper 在解码器中添加了 dropout。Dense 层合并了投影向量，并且 BahdanauAttention 与 AttentionWrapper 一起捕获了编码器输出和解码器之间的 attention。请注意，我们还使用不同的解码机制进行了训练和推理：

```
def training_decoding_layer(decoding_embed_input, en_len, decoding_cell,
initial_state, op_layer,             v_size, max_en_len):
    helper =
TrainingHelper(inputs=decoding_embed_input,sequence_length=en_len,time_majo
r=False)
    dec = BasicDecoder(decoding_cell,helper,initial_state,op_layer)
    logits, _, _ =
dynamic_decode(dec,output_time_major=False,impute_finished=True,
    maximum_iterations=max_en_len)
    return logits
```

训练中使用了 TensorFlow seq2seq 库中的常规 `TrainingHelper`，而在推理中则使用了 `GreedyEmbeddingHelper`：

```
def inference_decoding_layer(embeddings, start_token, end_token,
decoding_cell,
        initial_state, op_layer,max_en_len, batch_size):
    start_tokens = tf.tile(tf.constant([start_token], dtype=tf.int32,
                   [batch_size],name='start_tokens')
    inf_helper = GreedyEmbeddingHelper(embeddings,start_tokens,end_token)
    inf_decoder =
BasicDecoder(decoding_cell,inf_helper,initial_state,op_layer)
    inf_logits, _, _ =
dynamic_decode(inf_decoder,output_time_major=False,impute_finished=True,
                        maximum_iterations=max_en_len)
    return inf_logits
```

`GreedyEmbeddingHelper` 在编码器的输出投影向量中选择概率最大的单词。

- **序列到序列模型**

现在，我们要将编码器和解码器相结合以创建序列到序列模型：

```
def seq2seq_model(input_data, target_en_data, dropout_prob, fr_len, en_len,
max_en_len,
            v_size, rnn_cell_size, n_layers, word2int_en, batch_size):
    input_word_embeddings = tf.Variable(fr_embeddings_matrix,
name="input_word_embeddings")
    encoding_embed_input = tf.nn.embedding_lookup(input_word_embeddings,
input_data)
    encoding_op, encoding_st = encoding_layer(rnn_cell_size, fr_len,
                                n_layers, encoding_embed_input,
dropout_prob)
    decoding_input = process_encoding_input(target_en_data, word2int_en,
batch_size)
    decoding_embed_input = tf.nn.embedding_lookup(en_embeddings_matrix,
decoding_input)
    tr_logits, inf_logits = decoding_layer(decoding_embed_input,
en_embeddings_matrix,
                            encoding_op,encoding_st, v_size,
                            fr_len, en_len, max_en_len,
                            rnn_cell_size, word2int_en,
                            dropout_prob, batch_size,n_layers)
    return tr_logits, inf_logits
```

seq2seq_model 函数结合了源文本嵌入、编码器和解码器，当输入法语文本嵌入 fr_embeddings_matrix 时，可以输出 logits。编码器层和解码器层是使用先前定义的函数创建的。

- **建立图**

现在我们将结合先前所创建的所有组件以建立完整的图：

```
train_graph = tf.Graph()
with train_graph.as_default():
    input_data, targets, learning_rate, dropout_probs,
                            en_len, max_en_len, fr_len =model_inputs()
logits_tr, logits_inf = seq2seq_model(tf.reverse(input_data, [-1]),
targets, dropout_probs,
                        fr_len,en_len,max_en_len,
                        len(en_word2int)+1,rnn_len, n_layers,
                        en_word2int,batch_size)
logits_tr = tf.identity(logits_tr.rnn_output, 'logits_tr')
logits_inf = tf.identity(logits_inf.sample_id, name='predictions')
seq_masks = tf.sequence_mask(en_len, max_en_len, dtype=tf.float32,
name='masks')
with tf.name_scope("optimizer"):
    tr_cost = sequence_loss(logits_tr,targets,seq_masks)
    optimizer = tf.train.AdamOptimizer(learning_rate)
    gradients = optimizer.compute_gradients(tr_cost)
    capped_gradients = [(tf.clip_by_value(gradient, -5., 5.), var) for
gradient, var in gradients
                        if gradient is not None]
train_op = optimizer.apply_gradients(capped_gradients)
tf.summary.scalar("cost", tr_cost)
print("Graph created.")
```

首先使用 seq2seq_model 函数来创建带有 attention 的编码器–解码器网络，并使用 TensorFlow seq2seq 库中的 sequence_loss 函数及其输出 logits 值来计算损失。我们还在损失计算中屏蔽了填充。最终，使用 AdamOptimizer 作为损失优化器。

- **训练**

下面要在法语句子及其相应英语翻译上对网络展开训练。在此之前，我们将研究输出训练批次的函数：

```
def get_batches(en_text, fr_text, batch_size):
    for batch_idx in range(0, len(fr_text)//batch_size):
        start_idx = batch_idx * batch_size
        en_batch = en_text[start_idx:start_idx + batch_size]
        fr_batch = fr_text[start_idx:start_idx + batch_size]
        pad_en_batch = np.array(pad_sentences(en_batch, en_word2int))
        pad_fr_batch = np.array(pad_sentences(fr_batch,fr_word2int))
        pad_en_lens = []
        for en_b in pad_en_batch:
            pad_en_lens.append(len(en_b))
```

```
        pad_fr_lens = []
        for fr_b in pad_fr_batch:
            pad_fr_lens.append(len(fr_b))
        yield pad_en_batch, pad_fr_batch, pad_en_lens, pad_fr_lens
```

get_batches 函数将返回法语和英语句子的批次大小 batch_size。它还使用填充词元对句子进行填充，使得所有句子的长度都与批中的最大长度相等。现在我们来看一下训练循环：

```
min_learning_rate = 0.0006
display_step = 20
stop_early_count = 0
stop_early_max_count = 3
per_epoch = 3
update_loss = 0
batch_loss = 0
summary_update_loss = []
en_train = en_filtered[0:30000]
fr_train = fr_filtered[0:30000]

update_check = (len(fr_train)//batch_size//per_epoch)-1
checkpoint = logs_path + 'best_so_far_model.ckpt'
with tf.Session(graph=train_graph) as sess:
    tf_summary_writer = tf.summary.FileWriter(logs_path, graph=train_graph)
    merged_summary_op = tf.summary.merge_all()
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(1, epochs+1):
        update_loss = 0
        batch_loss = 0
        for batch_i, (en_batch, fr_batch, en_text_len, fr_text_len) in
enumerate(
                 get_batches(en_train, fr_train, batch_size)):
            before = time.time()
            _,loss,summary = sess.run([train_op,
tr_cost,merged_summary_op],
                            {input_data: fr_batch,
                            targets: en_batch,learning_rate: lr,
                            en_len: en_text_len,fr_len:
fr_text_len,dropout_probs: dr_prob})
            batch_loss += loss
            update_loss += loss
            after = time.time()
            batch_time = after - before
            tf_summary_writer.add_summary(summary, epoch_i * batch_size +
batch_i)
            if batch_i % display_step == 0 and batch_i &gt; 0:
                print('** Epoch {:&gt;3}/{} Batch {:&gt;4}/{} -
                    Batch Loss: {:&gt;6.3f}, seconds:
{:&gt;4.2f}'.format(epoch_i,epochs, batch_i,
                    len(fr_filtered) // batch_size, batch_loss /
display_step,
                    batch_time*display_step))
                batch_loss = 0
            if batch_i % update_check == 0 and batch_i &gt; 0:
```

```
                print("Average loss:", round(update_loss/update_check,3))
                summary_update_loss.append(update_loss)
                if update_loss &lt;= min(summary_update_loss):
                    print('Saving model')
                    stop_early_count = 0
                    saver = tf.train.Saver()
                    saver.save(sess, checkpoint)
                else:
                    print("No Improvement.")
                    stop_early_count += 1
                    if stop_early_count == stop_early_max_count:
                        break
                update_loss = 0
        if stop_early_count == stop_early_max_count:
            print("Stopping Training.")
            break
Output :

** Epoch   5/20 Batch  440/3131 - Batch Loss:  1.038, seconds: 170.97
** Epoch   5/20 Batch  460/3131 - Batch Loss:  1.154, seconds: 147.05
Average loss: 1.139
Saving model
```

以上代码的主要部分是训练循环。在该循环中，我们获取批次数据并将其馈送到网络，接着跟踪损失并在损失有所改善的情况下保存模型。如果 `stop_early_max_count` 的损失没有改善，训练过程将终止。最终我们发现平均损失从 6.49 降低至了 1.139 左右。



注意，损失值每次运行都可能发生变化。请参考 notebook 文件以获得完整输出。

- **推理**

我们将从检查点文件中载入模型，并在一系列示例数据上测试翻译效果：

```
with tf.Session(graph=loaded_graph) as sess:
    loader = tf.train.import_meta_graph(checkpoint + '.meta')
    loader.restore(sess, checkpoint)
    input_data = loaded_graph.get_tensor_by_name('input_data:0')
    logits = loaded_graph.get_tensor_by_name('predictions:0')
    fr_length = loaded_graph.get_tensor_by_name('fr_len:0')
    en_length = loaded_graph.get_tensor_by_name('en_len:0')
    dropout_prob = loaded_graph.get_tensor_by_name('dropout_probs:0')
    result_logits = sess.run(logits, {input_data: [fr_text]*batch_size,
                    en_length: [len(fr_text)],
                    fr_length: [len(fr_text)]*batch_size,
                    dropout_prob: 1.0})[0]

pad = en_word2int[TOKEN_PAD]
print('\nFrench Text')
print(' Word Ids: {}'.format([i for i in fr_text]))
print(' Input Words: {}'.format(" ".join( [fr_int2word[i] for i in fr_text
```

```
] )))
print('\nEnglish Text')
print(' Word Ids: {}'.format([i for i in result_logits if i != pad]))
print(' Response Words: {}'.format(" ".join( [en_int2word[i]for i in
result_logits if i!=pad] )))
print(' Ground Truth: {}'.format(" ".join( [en_int2word[i] for i in
en_filtered[random]] )))
```

接着我们将导入输入和输出预测张量以在测试数据上完成图的运行。以下是模型输出的部分翻译：

```
Unseen Test Data

French Text
Word Ids: [119, 67, 1003, 699, 11, 192, 13740]
Input Words: C'est environ 100 millions de ces planètes.
English Text
Word Ids: [119, 61, 1004, 2467, 21, 193, 17860]
Response Words: It's about 100 million of these planets.
Ground Truth: It's about 100 million such planets. &lt;EOS&gt;

French Text
Word Ids: [1255, 34, 21, 1263, 147, 1591, 609, 111, 1466, 3388, 21, 12253,
21, 22, 1673, 816]
Input Words: Qu'est-ce que les gens ont voulu donner au premier groupe, les
20% les plus pauvres ?
English Text
Word Ids: [320, 227, 227, 1511, 8, 636, 77, 14, 257, 2010, 14, 5433, 21,
14, 5433, 39610]
Response Words: What guys guys wanted to give at the first group, the
poorest of the poorest &lt;UNK&gt;
Ground Truth: What did people want to give to the first group, the bottom
20 &lt;UNK&gt; &lt;EOS&gt;INFO:tensorflow:Restoring parameters from
/tmp/models/best_so_far_model.ckpt

French Text
Word Ids: [31, 14982, 972, 33, 1043, 111, 2822, 131, 162, 4136, 11, 388,
94, 34, 7, 7499, 119, 413, 2973]
Input Words: La 2e ètape est d'apprendre au chien à avoir envie de faire ce
que vous voulez. C'est très simple.
English Text
Word Ids: [34, 1560, 989, 1, 793, 8, 96, 8, 391, 8, 391, 99, 132, 128,
2938, 39612]
Response Words: The second step is learning to have to do to do what you're
familiar simple. &lt;EOS&gt;
Ground Truth: So the second stage in training is to teach the dog to want
to do what we want him to do, and this is very easy. &lt;EOS&gt;

French Text
Word Ids: [722, 4957, 873, 974, 6186, 24, 718, 816, 19704, 125, 1001, 375,
977]
Input Words: J'ai rèpondu : « Bien, et pourquoi ? &lt;UNK&gt; un peu plus. »
English Text
```

```
Word Ids: [52, 153, 2773, 29, 744, 897, 1609, 136, 215, 3141, 1111, 1124,
39610]
Response Words: I said, "Well, and why what? let's a little bit more more.
&lt;UNK&gt;
Ground Truth: And I said, "Well, why? Tell me a little bit about it."
&lt;EOS&gt;
```

可以看出，尽管第一个示例短语在训练过程中对网络不可见，但其翻译仍与实际结果非常接近。我们还测试了部分训练数据：

```
Training Data

French Text
Word Ids: [422, 53, 1668, 277, 29, 378, 19704, 131, 1937, 4373, 4374, 218,
369, 538, 204, 157, 396, 704, 1791, 817, 34, 129, 87, 2172, 704, 1791, 87,
2172, 704, 3681]
Input Words: Donc pour moi, c'est une chose &lt;UNK&gt; à faire, d'essayer
d'atteindre l'autre côté avant qu'il ne soit trop tard, parce que quand il
sera trop tard, il sera trop tard.
English Text
Word Ids: [420, 57, 1323, 83, 1, 136, 464, 39610, 8, 391, 678, 8, 566, 225,
223, 564, 564, 118, 563, 737, 4316, 178, 131, 118, 565, 737, 4317, 39612]
Response Words: So for me, this is a thing &lt;UNK&gt; to do try to start
other side before before it doesn't too late, because when it will too
late. &lt;EOS&gt;
Ground Truth: So to me, this is the courageous thing to do, to try to reach
the other side before it's too late, because when it's going to be too
late, it's going to be too late. &lt;EOS&gt;

French Text
Word Ids: [1029, 33, 94, 625, 19704, 11, 3404, 735, 11, 19704, 131, 12,
5973, 816]
Input Words: Quel est ce besoin &lt;UNK&gt; de l'argent, puis de
&lt;UNK&gt; à la philanthropie ?
English Text
Word Ids: [2168, 83, 657, 21, 2853, 29, 319, 39610, 39610, 39612]
Response Words: What's this need of money, and then &lt;UNK&gt; &lt;UNK&gt;
&lt;EOS&gt;
Ground Truth: Why the need for accumulating money, then doing &lt;UNK&gt;
&lt;EOS&gt;
```



训练数据的翻译质量与测试数据并无不同。请注意，尽管仅使用了 50%的数据进行训练，但我们仍可以获得不错的翻译。你也可以在全部数据上进行训练以获得更好的结果。

- **TensorBoard 可视化**

我们将使用 TensorBoard 来对训练过程中的网络图和损失进行大致的观察，如图 10-4 所示。

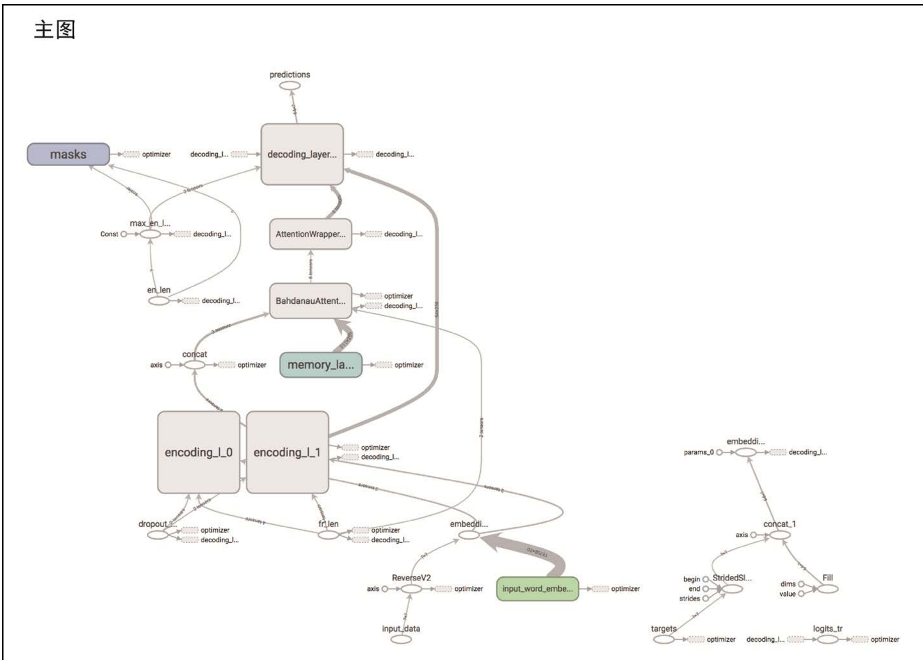



图 10-4　训练过程中的网络图及损失

可以看到，源文本和目标文本的对应词嵌入被馈送到了编码层和解码层中。attention 机制通过记忆层的权重来耦合编码器输出和解码器。你可以分别点击 TensorBoard 中的各个组件以了解其连接和张量尺寸的详细信息。现在来看一下数据训练期间的代价函数图，如图 10-5 所示。

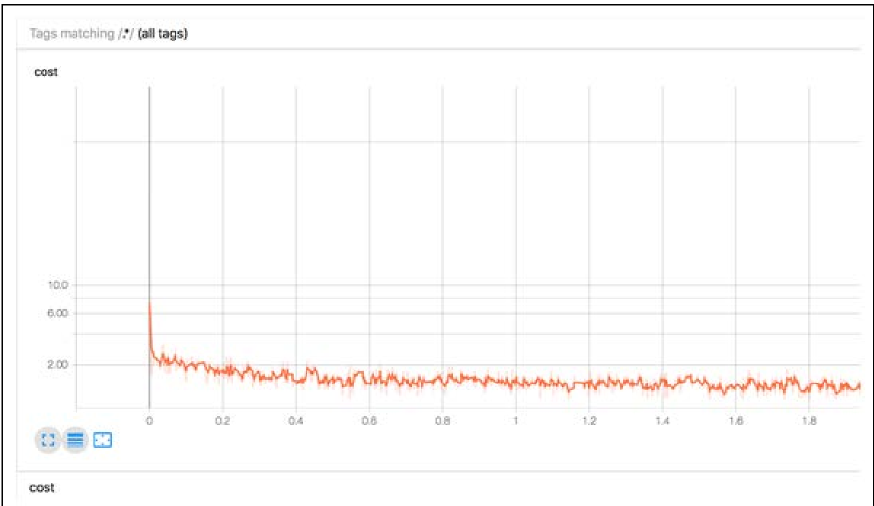



图 10-5　代价函数图

## 10.2　小结

本章描述了几种常用的机器翻译方法，并着重介绍了 NMT。我们在词法对齐模型的背景下简要描述了经典的 SMT，同时展示了使用 NLTK 构建 SMT 对齐模型的简单示例。当我们拥有大量双语数据时，可以使用该模型进行翻译。

这种模型的主要缺点在于不能很好地推广到除训练模型以外的领域（或背景）。近年来，深度神经网络已经成为机器翻译的流行方法，这主要是因为它们能有效产生接近人类水平的翻译。我们详细描述了如何使用 RNN 构建 NMT 模型，并用 TED 演讲的真实数据集进行了训练，最终将法语短语翻译成了英语。除了法语–英语翻译外，你还可以针对自身的机器翻译问题来使用类似的模型，也可以通过使用其他 attention 机制或更深层的编码器/解码器网络来改进本章所开发的模型。

第 11 章

# 使用 DeepSpeech 进行语音识别

语音识别指的是机器或计算机将口语转换为文本的任务。一个很好的示例是 Google 文档中的语音打字功能，该功能可以在你说话时实时将语音转换为文本。在本章中，我们将学习如何使用深度学习模型构建该系统。我们还将特别关注使用**循环神经网络**（RNN）模型，这类模型在语音识别实践中很有效果，因为它们可以捕捉语音数据中的时序依赖（这在语音转文本任务中很重要）。

本章涵盖以下主题：

- 语音识别概述
- 用于孤立单词识别的 RNN 模型
- 使用 DeepSpeech 模型进行连续语音的识别

## 11.1 语音识别概述

语音识别是一项复杂的任务，因为它必须考虑数据变化的几种来源，包括发声者的变化、词汇量大小、环境噪声、口音、发声者性格，等等。例如，一个人能很快说出像 apple 这样的单词，而另一个人却可能说得更慢一些。在两种情况下，语音识别系统都应产生单词 apple。语音识别的最常见方法是使用**隐马尔可夫模型**（HMM）。这是因为语音数据可以被视为随机或概率过程，并且对于一段很短的时间，可以认为切片与时间无关。我们可以在代表音素或单词的语音数据片段上训练 HMM 模型，然后用其预测可以组合生成语音文本的下一个音素或单词。

语音识别可以进一步分为孤立单词识别和连续语音识别。孤立单词识别是指将语音转录为清晰划定的单个单词，而连续语音识别是指我们通常连续说出单词的方式。显然，与后者相比，前者是一项更轻松的任务。孤立的单词通常有清晰的发音，并且其边界更容易识别，但连续语音在发音、间隔和单词间依存关系上可能会有很多变化。HMM 被用于处理这两种类型的任务已有一段时间了，该模型在语音数据中采用特定的结构，并且可以在较少的数据上进行训练。使用

BiLSTM 的最新模型可以执行端到端的语音识别，且不对数据结构做要求。该网络可以学习数据中的特征并捕获时序依赖性。一个简单的规则是：如果我们有更多数据，则使用神经网络模型可能会比使用 HMM 产生更好的结果。接下来，我们将研究孤立单词识别的简单模型。

## 11.2 建立用于语音识别的 RNN 模型

我们会将来自 Jakobovski 的免费数字音频数据集用于我们的基本模型。请将数据下载到计算机系统上的任意目录中。在示例代码中，使用数据被复制到的路径替换指定.wav 文件的路径。



注意，我们将数据分为包括 1470 个文件的训练集和包括 30 个文件的测试集。

在深入探讨模型的细节之前，我们将研究如何为训练做好准备。实际上，最常用的预处理步骤是将原始音频数据转换为其频谱。频谱或功率谱类似于数据的指纹，原始音频在其中被分解为一个个部分或频率。与其他表示方法相比，频谱表示有助于确定信号中的哪些频率（高音调或低音调）占据主导地位（在功率或能量方面）。现在，我们将研究如何提取这种频谱表示或功率谱表示。

### 11.2.1 语音信号表示

现在来看一下如何从语音数字信号数据集中提取频谱。该数据集包含以.wav 文件形式记录的语音记录。我们将利用常用于音频数据分析的 librosa 库。 首先使用以下命令安装软件包：

```
pip install librosa
```

关于安装 librosa 库的其他方法，请查看其主页。我们将使用音频信号的**梅尔频率倒谱系数**（Mel frequency cepstral coefficient）特征，该特征是从信号的短时帧中获得的一种功率谱，其主要假设是：对于 20 ~ 40 毫秒量级的短持续时间，频谱变化不大。因此，将信号切片成这些短时间段，并为每个切片计算频谱。幸运的是，我们不必深入这些细节，因为 librosa 库可以为我们做到这一点。利用以下函数提取 MFCC 特征：

```
def get_mfcc_features(fpath):
    raw_w,sampling_rate = librosa.load(fpath,mono=True)
    mfcc_features = librosa.feature.mfcc(raw_w,sampling_rate)
    if(mfcc_features.shape[1]>utterance_length):
        mfcc_features = mfcc_features[:,0:utterance_length]
    else:
        mfcc_features=np.pad(mfcc_features,((0,0),(0,utterance_length-mfcc_
features.shape[1])),
                          mode='constant', constant_values=0)
    return mfcc_features
```

`librosa.load` 函数加载.wav 文件并输出原始 wav 数据 `raw_w` 及其采样率 `sample_rate`。

MFCC 特征则是通过对原始数据调用 `librosa.feature.mfcc` 函数获得的。请注意，我们还将特征尺寸截断为 `utterance_length`（代码中将其设置为 `35`），该参数是根据数字数据集中发声的平均长度设置的。如果需要的话，也可以尝试使用更高的值。要了解更多信息，请查看本书代码库中的 Chapter11/01_example.ipynb。现在，我们将打印特征的维度并将其绘制出来，以查看功率谱：

```
import matplotlib.pyplot as plt
import librosa.display
%matplotlib inline
mfcc_features =
get_mfcc_features('../../speech_dset/recordings/train/5_theo_45.wav')
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
librosa.display.specshow(mfcc_features, x_axis='time')
print("Feature shape: ", mfcc_features.shape)
print("Features: ", mfcc_features[:,0])
```

```
Output:
Feature shape: (20, 35)
Features:[-5.16464322e+02 2.18720111e+02 -9.43628435e+01 1.63510496e+01
  2.09937445e+01 -4.38791200e+01 1.94267052e+01 -9.41531735e-02
 -2.99960992e+01 1.39727129e+01 6.60561909e-01 -1.14758965e+01
  3.13688180e+00 -1.34556070e+01 -1.43686686e+00 1.17119580e+01
 -1.54499037e+01 -1.13105764e+01 2.53027299e+00 -1.35725427e+01]
```

可以看到，数字 5 的语音信号频谱中用于音频信号的 35 个时间片由 20 个特征（librosa 默认值）组成。我们还可以看到第一个时间片的 MFCC 特征值。现在来可视化 MFCC 特征，如图 11-1 所示。

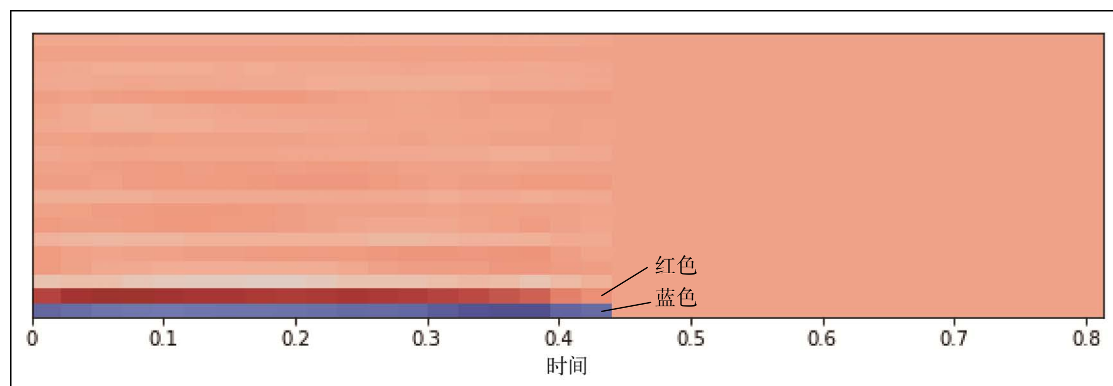



图　11-1

图 11-1 中越红（深灰色）的区域表示 MFCC 系数的值越大，而越蓝（浅灰色）的区域表示 MFCC 系数的值越小。现在，我们将建立一个简单的模型来识别音频数据中的数字。

### 11.2.2 用于语音数字识别的LSTM模型

为简便起见，本例将使用 tflearn 包。该包可以通过以下命令安装：

```
pip install tflearn
```

我们将定义用于读入.wav 文件并对其进行预处理以便用于批训练的函数：

```
def get_batch_mfcc(fpath,batch_size=256):
    ft_batch = []
    labels_batch = []
    files = os.listdir(fpath)
    while True:
        print("Total %d files" % len(files))
        random.shuffle(files)
        for fname in files:
            if not fname.endswith(".wav"):
                continue
    mfcc_features = get_mfcc_features(fpath+fname)
    label = np.eye(10)[int(fname[0])]
    labels_batch.append(label)
    ft_batch.append(mfcc_features)
    if len(ft_batch) >= batch_size:
        yield ft_batch, labels_batch
    ft_batch = []
    labels_batch = []
```

在 `get_batch_mfcc` 函数中，我们读取.wav 文件并使用之前定义的 `get_mfcc_features` 函数提取 MFCC 特征。标签是从 0 到 9 的 10 位独热键编码。然后该函数默认以 256 的批次大小返回数据。接下来定义 LSTM 模型：

```
train_batch = get_batch_mfcc('../../speech_dset/recordings/train/')
sp_network = tflearn.input_data([None, audio_features, utterance_length])
sp_network = tflearn.lstm(sp_network, 128*4, dropout=0.5)
sp_network = tflearn.fully_connected(sp_network, ndigits,
activation='softmax')
sp_network = tflearn.regression(sp_network, optimizer='adam',
learning_rate=lr, loss='categorical_crossentropy')
sp_model = tflearn.DNN(sp_network, tensorboard_verbose=0)
while iterations_train > 0:
    X_tr, y_tr = next(train_batch)
    X_test, y_test = next(train_batch)
    sp_model.fit(X_tr, y_tr, n_epoch=10, validation_set=(X_test, y_test),
show_metric=True, batch_size=bsize)
    iterations_train-=1
sp_model.save("/tmp/speech_recognition.lstm")
```

该模型基本上由一个 LSTM 层和一个全连接层组成。我们使用分类交叉熵作为 Adam 优化器的损失函数，并使用 `get_batch_mfcc` 函数的批处理输入来训练模型。300 个周期后，可得到以下输出：

```
Training Step: 1199 | total loss: 0.45749 | time: 0.617s
| Adam | epoch: 300 | loss: 0.45749 - acc: 0.8975 -- iter: 192/256
```

接下来，我们对测试集中的语音文件进行预测。被试音频来自于数字 4 的语音信号：

```
sp_model.load('/tmp/speech_recognition.lstm')
mfcc_features =
get_mfcc_features('../../speech_dset/recordings/test/4_jackson_40.wav')
mfcc_features =
mfcc_features.reshape((1,mfcc_features.shape[0],mfcc_features.shape[1]))
prediction_digit = sp_model.predict(mfcc_features)
print(prediction_digit)
print("Digit predicted: ", np.argmax(prediction_digit))


Output:
INFO:tensorflow:Restoring parameters from /tmp/speech_recognition.lstm
[[2.3709694e-03 5.1581711e-03 7.8898791e-04 1.9530311e-03 9.8459840e-01
  1.1394228e-03 3.0317350e-04 1.8992715e-03 1.6027489e-03 1.8592674e-
  04]]
Digit predicted: 4
```

我们加载训练后的模型，并获得测试音频文件的特征。从输出可以看出，该模型能够预测出正确的数字。我们还可以分别看到 10 个数字的预测概率。接下来，我们将研究 TensorBoard 中的模型可视化。

## 11.2.3 TensorBoard可视化

首先看看训练过程中准确率和损失的变化。打开 TensorBoard，指向日志目录/tmp/tflearn_logs（tflearn 默认），如图 11-2 所示。

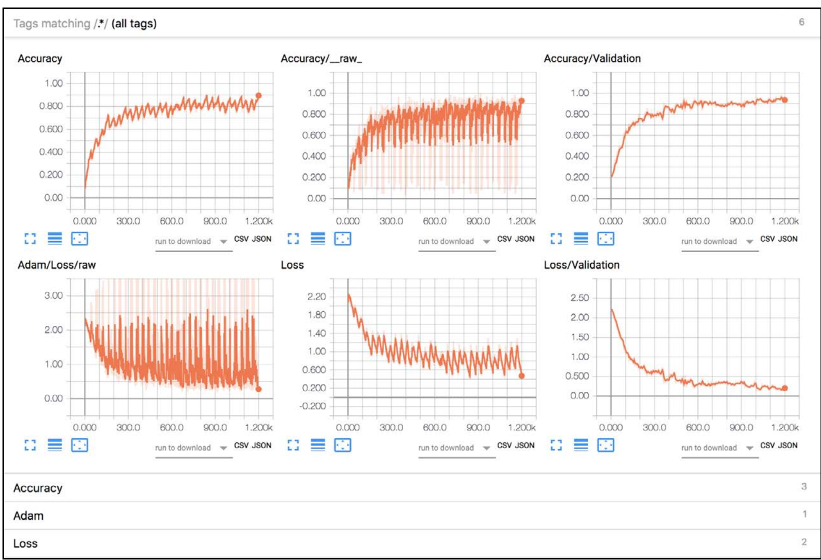



图 11-2

我们发现验证损失和训练损失都随时间步长而减少。请注意，此处使用的验证集也来自于训练集。这只是一个便捷的小技巧。你可以为原始数据保留一个单独的验证集，就像为测试集一样。接下来，我们将在 TensorBoard 中查看模型的图，如图 11-3 所示。

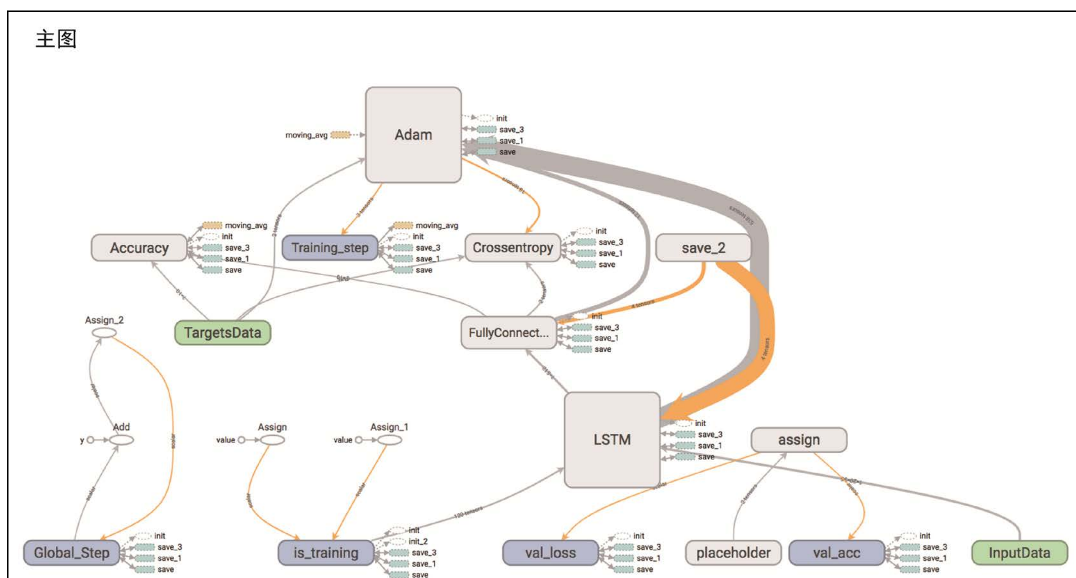



图 11-3 TensorBoard 中的模型图

如前所述，我们通过一个输入层将 MFCC 音频特征张量馈送到 LSTM 层，而 LSTM 的输出被馈送到用于输出预测的全连接层。接下来，我们将研究如何使用 DeepSpeech 架构创建语音转文本模型。

### 11.2.4 使用DeepSpeech架构的语音转文本模型

DeepSpeech 是一个端到端的架构，其中，深度学习取代了传统手工设计的语音转文本算法。该模型运行良好，不受说话人影响，因为它可以直接从数据中学习。我们将简要介绍 DeepSpeech 的模型架构。

#### 1. DeepSpeech 模型概述

该模型由一堆全连接隐藏层、一个双向 RNN 和输出部分的其他隐藏层组成。前三个非递归层的作用类似于 RNN 层的预处理步骤。一种附加做法是使用 ReLU 以防止激活爆炸。输入音频特征是非递归层在频谱图的时间切片中所看到的梅尔倒谱系数。除了通常的时间切片外，模型还对频谱数据进行了预处理，以包括过去和将来的语境。第四层是 RNN 层，具有正向递归和反向递归。第五层采用向前和向后递归的串联输出，产生输出并将其馈送到预测字符概率的最终

softmax 层。图 11-4 展示了原始论文中的模型架构，其中蓝色（灰色水平实线）和红色（灰色水平虚线）箭头分别表示隐藏层和双向递归层。

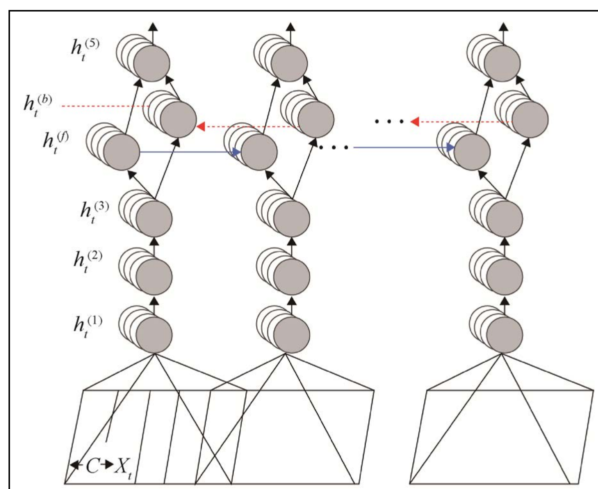



图　11-4

带有上下文的音频输入也与时间切片的 MFCC 功能特征一起显示了出来。现在，我们将研究如何在 TensorFlow 中实现该模型。完整的 Jupyter Notebook 可在本书代码库中找到（Chapter11/02_example.ipynb）。在此之前，我们将简要介绍用于模型训练的数据。

### 2. 语音录音数据集

我们将利用 Kaggle 上**语言数据协会**（Linguistic Data Consortium，LDC）的语音录音。你可以使用 Kaggle 账户下载该数据集。该数据包括不同说话者的自由语音录音。尽管原始数据集非常大（有几吉字节），但 Kaggle 中的数据只是它很小的一个子集，因此我们可以在合理的时间内进行训练。请注意，语音转文本的工作可能需要大量的语音转录数据，并可能需要花费数小时或数天的时间才能训练得到一个良好、有意义转录效果的模型。（当然）你也可以在较大数据上构建相同模型，以实现准确率更好的语音转文本。有关本节中的完整代码，可以参考本书代码库中的 Jupyter Notebook（Chapter11/02_example.ipynb）。

### 3. 预处理语音数据

就像先前的示例一样，我们从音频数据中提取 MFCC 特征。除此之外，我们还添加了原始论文中使用的上下文：

```
def audiofile_to_vector(audio_fname, n_mfcc_features, nctx):
    sampling_rate, raw_w = wavfile.read(audio_fname)
    mfcc_ft = mfcc(raw_w, samplerate=sampling_rate, numcep=n_mfcc_features)
    mfcc_ft = mfcc_ft[::2]
```

```
    n_strides = len(mfcc_ft)
    dummy_ctx = np.zeros((nctx, n_mfcc_features), dtype=mfcc_ft.dtype)
    mfcc_ft = np.concatenate((dummy_ctx, mfcc_ft, dummy_ctx))
    w_size = 2*nctx+1
    input_vector = np.lib.stride_tricks.as_strided(mfcc_ft,(n_strides,
w_size,
                    n_mfcc_features,(mfcc_ft.strides[0],
mfcc_ft.strides[0], mfcc_ft.strides[1]),
                    writeable=False)
    input_vector = np.reshape(input_vector, [n_strides, -1])
    input_vector = np.copy(input_vector)
    input_vector = (input_vector -
np.mean(input_vector))/np.std(input_vector)
    return input_vector
```

首先读取.wav 文件，然后提取 MFCC 特征 `mfcc_ft`，并使用 NumPy `as_strided` 函数将模拟上下文添加到每个时间切片的前后。`audiofile_to_vector` 函数最终返回带有过去和将来上下文的 MFCC 特征。接下来，我们将研究如何从数据集中提取源文本和转录文本以进行批处理训练：

```
def get_wav_trans(fpath,X, y):
    files = os.listdir(fpath)
    for fname in files:
        next_path = fpath + "/" + fname
        if os.path.isdir(next_path):
            get_wav_trans(next_path,X,y)
        else:
        if fname.endswith('wav'):
            fname_without_ext = fname.split(".")[0]
            trans_fname = fname_without_ext + ".txt"
            trans_fname_path = fpath + "/" + trans_fname
            if os.path.isfile(trans_fname_path):
                mfcc_ft = audiofile_to_vector(next_path,n_inp,n_ctx)
                with open(trans_fname_path,'r') as content:
                    transcript = content.read()
                    transcript = re.sub(regexp_alphabets, ' ',
transcript).strip().lower()
            trans_lbl = get_string2label(transcript)
        X.append(mfcc_ft)
        y.append(trans_lbl)
```

我们通过提供的路径 `fpath` 递归提取所有 wav 文件的 MFCC 特征。`audio_file_to_vector` 函数（此前展示过）将提取我们找到的每个 wav 文件的 MFCC 特征，并从文本文件中读取相应的转录文本。同时，我们利用正则表达式 `regexp_alphabets` 从转录的文本中删除非字母字符。原始文本记录将被传递到 `get_string2label` 函数并被转换为整数标签用作模型训练的目标标签：

```
regexp_alphabets = "[^a-zA-Z']+"
cnt=0
def get_label(ch):
```

```
        global cnt
        label = cnt
        cnt+=1
        return label
    chr2lbl = {c:get_label(c) for c in list(chars)}
    lbl2chr = {chr2lbl[c]:c for c in list(chars)}
    def get_string2label(strval):
        strval = strval.lower()
        idlist = []
        for c in list(strval):
            if c in chr2lbl:
                idlist.append(chr2lbl[c])
        return np.array(idlist)

    def get_label2string(lblarr):
        strval = []
        for idv in lblarr:
            strval.append(lbl2chr[idv])
        return ''.join(strval)
```

get_string2label 使用 chr2lbl 字典将字符串转换为整数标签列表，该字典将 a-z 的字符（另外带有空格和撇号）映射为整数值。同样，我们使用 get_label2string 函数将标签列表（通过反向映射 lbl2chr）转换为原始字符串。接下来，我们将研究如何创建 DeepSpeech 模型。

#### 4. 创建模型

我们将准确复制 DeepSpeech 原始论文中描述的模型。如前所述，该模型由循环层和非循环层组成。现在在代码中查看 get_layers 函数：

```
    with tf.name_scope('Lyr1'):
        B1 = tf.get_variable(name='B1', shape=[n_h],
        initializer=tf.random_normal_initializer(stddev=0.046875))
        H1 = tf.get_variable(name='H1', shape=[n_inp + 2*n_inp*n_ctx, n_h],
        initializer=tf.contrib.layers.xavier_initializer(uniform=False))
        logits1 = tf.add(tf.matmul(X_batch, H1), B1)
        relu1 = tf.nn.relu(logits1)
        clipped_relu1 = tf.minimum(relu1,20.0)
        Lyr1 = tf.nn.dropout(clipped_relu1, 0.5)

    with tf.name_scope('Lyr2'):
        B2 = tf.get_variable(name='B2', shape=[n_h],
        initializer=tf.random_normal_initializer(stddev=0.046875))
        H2 = tf.get_variable(name='H2', shape=[n_h,n_h],
        initializer=tf.random_normal_initializer(stddev=0.046875))
        logits2 = tf.add(tf.matmul(Lyr1, H2), B2)
        relu2 = tf.nn.relu(logits2)
        clipped_relu2 = tf.minimum(relu2,20.0)
        Lyr2 = tf.nn.dropout(clipped_relu2, 0.5)

    with tf.name_scope('Lyr3'):
        B3 = tf.get_variable(name='B3', shape=[2*n_h],
        initializer=tf.random_normal_initializer(stddev=0.046875))
```

```
    H3 = tf.get_variable(name='H3', shape=[n_h,2*n_h],
    initializer=tf.random_normal_initializer(stddev=0.046875))
    logits3 = tf.add(tf.matmul(Lyr2, H3), B3)
    relu3 = tf.nn.relu(logits3)
    clipped_relu3 = tf.minimum(relu3,20.0)
    Lyr3 = tf.nn.dropout(clipped_relu3, 0.5)
```

前三个隐藏层是非循环的，其中参数 H1、H2、H3 和 B1、B2、B3 分别是各层的权重和偏差。各层的输出通过 ReLU 裁剪函数以避免梯度爆炸。同时我们还为前三个隐藏层提供了随机失活。请注意，第一个隐藏层权重的维度为[n_inp + 2 * n_ip * n_ctx, n_h]，这与带有上下文的输入 MFCC 相同。在下列代码中，我们将隐藏单元的数量 n_h 设置为 1024，将 MFCC 特征 n_inp 设置为 26，并将上下文 n_ctx 设置为 9。随后查看循环层：

```
with tf.name_scope('RNN_Lyr'):
    fw_c = tf.contrib.rnn.BasicLSTMCell(n_h, forget_bias=1.0,
state_is_tuple=True,
    reuse=tf.get_variable_scope().reuse)
    fw_c = tf.contrib.rnn.DropoutWrapper(fw_c, input_keep_prob=0.7,
            output_keep_prob=0.7,seed=123)
    bw_c = tf.contrib.rnn.BasicLSTMCell(n_h, forget_bias=1.0,
state_is_tuple=True,
    reuse=tf.get_variable_scope().reuse)
    bw_c = tf.contrib.rnn.DropoutWrapper(bw_c,input_keep_prob=0.7,
            output_keep_prob=0.7, seed=123)
    Lyr3 = tf.reshape(Lyr3, [-1, X_batch_shape[0], 2*n_h])
    outs, out_states = tf.nn.bidirectional_dynamic_rnn(cell_fw=fw_c,
                cell_bw=bw_c,inputs=Lyr3,dtype=tf.float32,time_major=True,
                sequence_length=seq_len)
    outs = tf.concat(outs, 2)
    outs = tf.reshape(outs, [-1, 2 * n_h])
```

如前所述，循环层是带有随机失活机制的双向 LSTM。向前和向后 LSTM 的级联输出被输入到下一个隐藏层。现在，我们将在输出中查看这两个隐藏层：

```
with tf.name_scope('Lyr4'):
    B4 = tf.get_variable(name='B4', shape=[n_h],
    initializer=tf.random_normal_initializer(stddev=0.046875))
    H4 = tf.get_variable(name='H4', shape=[(2 * n_h), n_h],
    initializer=tf.random_normal_initializer(stddev=0.046875))
    logits4 = tf.add(tf.matmul(outs, H4), B4)
    relu4 = tf.nn.relu(logits4)
    clipped_relu4 = tf.minimum(relu4,20.0)
    Lyr4 = tf.nn.dropout(clipped_relu4, 0.5)

with tf.name_scope('Lyr5'):
    B5 = tf.get_variable(name='B5', shape=[n_h],
    initializer=tf.random_normal_initializer(stddev=0.046875))
    H5 = tf.get_variable(name='H5', shape=[n_h, n_chars],
    initializer=tf.random_normal_initializer(stddev=0.046875))
    Lyr5 = tf.add(tf.matmul(Lyr4, H5), B5)
    Lyr5 = tf.reshape(Lyr5, [-1, X_batch_shape[0], n_chars])
```

同输入中的隐藏层一样，最后两个隐藏层分别具有权重 H4、H5 和偏差 B4、B5。第 4 层既带有 ReLU 激活，也配有随机失活机制。第五层将以一个字符为单位输出 n_chars（包括字母数及空白数）个字符的概率。之后，我们将研究损失函数和优化器的定义：

```
def get_cost(tgts,logits,len_seq):
    loss_t = ops.ctc_ops.ctc_loss(tgts, logits, len_seq)
    loss_avg = tf.reduce_mean(loss_t)
    return loss_avg

def get_optimizer(logits,len_seq,loss_avg):
    adm_opt =
tf.train.AdamOptimizer(learning_rate=plr,beta1=pb1,beta2=pb2,epsilon=peps)
    adm_opt = adm_opt.minimize(loss_avg)
    dec, prob_log = ops.ctc_ops.ctc_beam_search_decoder(logits, len_seq,
merge_repeated=False)
    return adm_opt,dec
```

我们将使用 TensorFlow tensorflow.python.ctc_ops.ctc_loss 中的**联结主义时间分类**（Connectionist Temporal Classification，CTC）损失函数。该函数以对数和目标变量作为输入并计算损失。在此，平均损失由 get_costs 函数计算，并使用 get_optimizer 函数中的 AdamOptimizer 最小化。现在，我们将看看用于模型训练的批数据馈送器：

```
class Batch:
    def __init__(self):
        self.start_idx = 0
        self.batch_size = 10
        self.audio = []
        self.transcript = []
get_wav_trans("../../speech_dset/timit/",self.audio,self.transcript)

    def pad_seq(self,seqs):
        seq_lens = np.asarray([len(st) for st in seqs], dtype=np.int64)
        n_s = len(seqs)
        max_seq_len = np.max(seq_lens)
        s_shape = tuple()
        for s in seqs:
            if len(s) > 0:
                s_shape = np.asarray(s).shape[1:]
                break
            seqs_trc = (np.ones((n_s, max_seq_len) + s_shape) *
0.).astype(np.float32)
            for ix, s in enumerate(seqs):
                if len(s) == 0:
                    continue
            trc = s[:max_seq_len]
            trc = np.asarray(trc, dtype=np.int64)
            if trc.shape[1:] != s_shape:
                raise ValueError("ERROR in truncation shape")
            seqs_trc[ix, :len(trc)] = trc
        return seqs_trc, seq_lens
```

```
    def get_sp_tuple(self,seqs):
        ixs = []
        vals = []
        for n, s in enumerate(seqs):
            ixs.extend(zip([n] * len(s), range(len(s))))
            vals.extend(s)
            ixs = np.asarray(ixs, dtype=np.int64)
            vals = np.asarray(vals, dtype=np.int32)
            shape = np.asarray([len(seqs), ixs.max(0)[1] + 1],
dtype=np.int64)
        return ixs, vals, shape

    def get_next_batch(self):
        src = self.audio[self.start_idx:self.start_idx+self.batch_size]
        tgt =
self.transcript[self.start_idx:self.start_idx+self.batch_size]
        self.start_idx += self.batch_size
        if(self.start_idx>len(self.audio)):
            self.start_idx=0
            src,src_len = self.pad_seq(src)
            sp_lbls = self.get_sp_tuple(tgt)
        return src, src_len, sp_lbls
```

我们利用 `get_wav_trans` 从.wav 和.txt 文件中获取 MFCC 特征 `audio` 和文本 `transcript`。`get_next_batch` 函数以 `batch_size` 的大小返回源（音频）和目标（文本），而 `pad_seq` 函数将 MFCC 序列填充到特定批序列的最大长度。类似地，`get_sp_tuple` 可以用来获取目标标签的稀疏表示。现在，我们来看一下训练参数设置：

```
def get_model():
    input_t = tf.placeholder(tf.float32, [None, None, n_inp + (2 * n_inp *
n_ctx)], name='inp')
    tgts = tf.sparse_placeholder(tf.int32, name='tgts')
    len_seq = tf.placeholder(tf.int32, [None], name='len_seq')
    logits = get_logits(input_t,tf.to_int64(len_seq))
    return input_t, tgts, len_seq, logits
```

`get_model` 函数为源（MFCC 特征）、目标（转录文本标签）和序列长度创建输入占位符张量，然后调用 `get_logits` 函数，而后者则调用前面所述的 `get_layers`。此函数可以创建模型。现在我们来看一下模型训练循环：

```
gr = tf.Graph()
with gr.as_default():
    input_t,tgts,len_seq,logits = get_model()
    loss_avg = get_cost(tgts,logits,len_seq)
    adm_opt, dec = get_optimizer(logits,len_seq,loss_avg)
    error_rate = get_error_rates(dec,tgts)
    sess = tf.Session()
    writer = tf.summary.FileWriter('/tmp/models/', graph=sess.graph)
    loss_summary = tf.summary.scalar("loss_avg", loss_avg)
    sum_op = tf.summary.merge_all()
    init_op = tf.global_variables_initializer()
    sess.run(init_op)
```

```
    for ep in range(epochs):
        train_cost = 0
        label_err_rate = 0
        batch_feeder = Batch()
        n_batches =
np.ceil(len(batch_feeder.audio)/batch_feeder.batch_size)
        n_batches = int(n_batches)
        st = time.time()
        for batch in range(n_batches):
            src,len_src,labels_src = batch_feeder.get_next_batch()
            data_dict = {input_t: src, tgts: labels_src,len_seq:len_src}
            batch_cost, _,summ = sess.run([loss_avg, adm_opt,sum_op],
data_dict)
            train_cost += batch_cost * batch_feeder.batch_size
            print("Batch cost: {0}, Train cost:
{1}".format(batch_cost,train_cost))
            label_err_rate += sess.run(error_rate, feed_dict=data_dict) *
batch_feeder.batch_size
            print('Label error: {}'.format(label_err_rate))
            writer.add_summary(summ,ep*batch_feeder.batch_size+batch)
            saver = tf.train.Saver()
            saver.save(sess, '/tmp/models/speech2txt.ckpt')
        decoded_val = sess.run(dec[0], feed_dict=data_dict)
        d_decoded_val = tf.sparse_tensor_to_dense(decoded_val,
                        default_value=-1).eval(session=sess)
        d_lbl = decoded_val_to_text(labels_src)
        cnt = 0
        cnt_max = 4
        if cnt < cnt_max:
            for actual_val, decoded_val in zip(d_lbl, d_decoded_val):
                d_str = array2txt(decoded_val)
                print('Batch {}'.format(batch))
                print('Actual: {}'.format(actual_val))
                print('Predicted: {}'.format(d_str))
                cnt += 1
        time_taken = time.time() - st
        log = 'Epoch {}/{}, training_cost: {:.3f}, error_rate: {:.3f},
time: {:.2f} sec'
        print(log.format(ep,epochs,train_cost/len(batch_feeder.audio),
            (label_err_rate/len(batch_feeder.audio)), time_taken))
```

我们首先通过调用 `get_model`、`get_cost`、`get_optimizer` 和 `get_error_rates`，分别创建图、损失函数、优化器并计算错误率以完成对图的设置和训练。我们初始化 `batch_feeder` 以批量获取训练数据，同时在字典 `feed` 中填充源 `src`、目标 `labels_src` 和源长度 `src_len`。在每个批次完成后，我们将打印该批次的误差、标签错误率并保存模型。在每个周期完成之后，我们还将打印示例预测文本。

### 5. TensorBoard 可视化

现在，我们将使用 TensorBoard 查看图形和损失函数。启动 TensorBoard 并指向代码中的日志目录（/tmp/models 或设置的其他路径）。该图的可视化结果如图 11-5 所示。

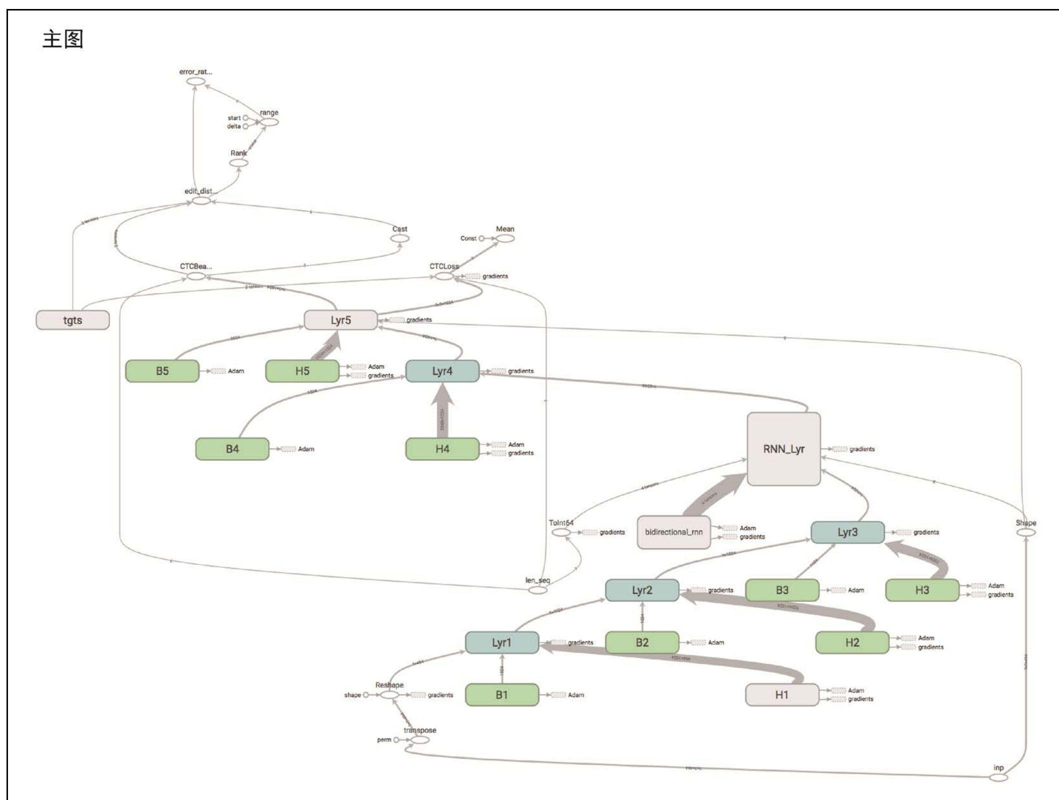



图 11-5　TensorBoard 上的模型图可视化

我们可以快速找到该模型与原始 DeepSpeech 论文中模型架构之间的相似之处：输入为非循环层，然后是 RNN 层和输出隐藏层。现在，我们将研究平均 CTC 损失随训练的变化情况，如图 11-6 所示。

可以看出，随着训练的进行，CTC 的损失稳步减少。尽管我们的模型复制了 DeepSpeech 架构，但仅在较小数据集上进行了训练。为了获得良好的转录准确度（单词错误率和 CRT 丢失等，这些与说话者无关），我们可能需要在大型数据集上展开训练。想了解详细信息，你还可以查看 GitHub 上 Mozilla 的另一个 DeepSpeech 实现。该实现使用了数十吉字节的数据集，并在多个 GPU 上进行了分布式训练。与之不同，我们这里展示的实现是为了快速提出一个简单的模型并进行训练。

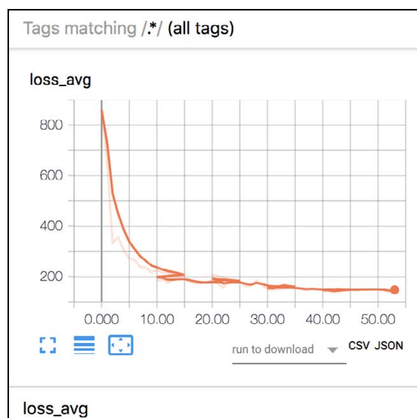



图　11-6

### 11.2.5　语音识别最新技术

原始的 DeepSpeech 架构使用语言模型来纠正字符序列中的某些错误，结合了 RNN 的输出和语言模型以得出给定语音录音最可能对应的文本序列。基于 attention 的方法由于能够有效提高非 attention 机制模型的准确率而得到流行，甚至已被用于语音识别。Google 的论文“Listen, Attend, and Spell: A Neural Network for Large Vocabulary Conversational Speech Recognition”就结合了 attention 机制来转录语音记录。

该系统的主要优点是使用基于 attention 的模型，并且不假定序列中的字符独立性（包括 DeepSpeech 在内的许多基于 CTC 的方法都会假定序列中后续字符之间的概率独立分布）。让我们简要看一下其架构，如图 11-7 所示。

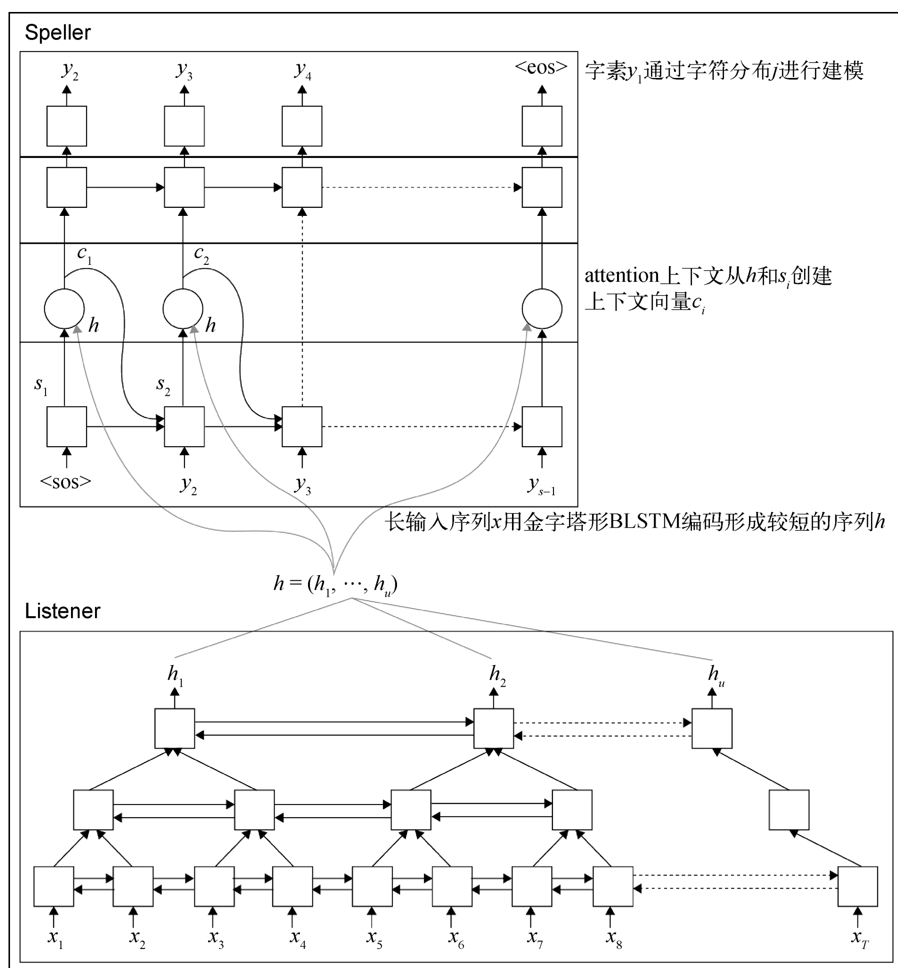



图　11-7

图 11-7 显示了该模型的主要组件，即 Listener 和 Speller。Listener 将输入音频频谱 $x$ 编码为高级特征表示 $h$。普通 BiLSTM 和 Listener 体系结构之间的主要区别在于前者使用了金字塔结构。这种结构降低了每个级别上非常大的输入音频序列的时间分辨率，从而减少了 Speller 必须参与以产生转录的有用音频信息的数量。Speller 使用 Listener 输出的最终高阶表示形式来计算 attention 上下文向量。然后，在每个时间步中，Speller 都将根据使用 attention 向量之前看到的所有重要字符来计算下一个字符的概率。作者指出，与神经网络和基于 HMM 的最新模型相比，该模型在**字错误率**（WER）方面准确率较高。

## 11.3　小结

本章描述了语音识别中的深度学习方法，概述了当前实际使用的语音识别软件。我们看到，基于 HMM 的传统方法可能需要与特定的语言模型相结合，而基于神经网络的方法可以完全从数据中学习端到端的语音转录。这是神经网络模型相较于 HMM 模型的主要优势之一。我们使用 TensorFlow 开发了基本的语音数字识别模型，然后使用开放的语音数字数据集对测试集进行训练并做出预测。该示例提供了语音识别系统所涉及任务的背景，例如从原始音频数据中提取频谱（如 MFCC 特征）并将文本转录本转换为标签。然后，我们介绍了百度的 DeepSpeech 架构，它是转录语音的一个流行模型。然后，我们解释了 TensorFlow 中对 DeepSpeech 模型的完整实现，并在 LDC 数据集的子集上进行了训练。为了进一步探索，读者可以调整模型参数并尝试更大的数据集。

然后，我们针对基于 attention 的模型简要介绍了语音识别的最新技术。我们研究了“Listen, Attend, and Spell”（LAS）论文中描述的基于 attention 的模型。尽管基于 CTC 的模型假定各个字符独立，但是 LAS 模型并未做出这样的假设。正如作者所述，这正是 LAS 优于类 DeepSpeech 模型的主要优势之一。有兴趣的读者可以在 GitHub 上查看该模型的 PyTorch 实现。

在下一章，我们将查看语音识别的逆任务，即文本转语音。

第 12 章

# 使用 Tacotron 进行文本转语音

文字转语音（text-to-speech，TTS）是将文字转换为可理解的自然语音的行为。在深入研究用于处理 TTS 的深度学习方法前，我们应当问自己以下问题：TTS 系统的用途是什么？我们为什么对其有需求呢？

TTS 有很多用途，最显而易见的一个用途是让盲人听到书面内容。实际上，并不总是有基于盲文的图书、设备和指示牌，也并不总是有人读给盲人听。在不久的将来，智能眼镜可能会被用来描述周围的环境，并为用户阅读城市指示牌和文本指引。

许多人从小就在学习障碍（如阅读障碍）中挣扎，而强大的 TTS 系统可以随时为其提供帮助，例如提高他们在学习或工作中的效率。

此外，在学习领域，不同的个体偏好不同的知识吸收方式。例如，有些人具有很好的视觉记忆力，另一些人更容易记住听到的信息，还有些人更依赖于他们的动觉记忆（与身体运动有关的记忆）。TTS 系统可以帮助听觉学习者利用这种独特的学习方式。

如今的世界日新月异，多任务处理已成为必需。我们经常看到有人在街上行走的同时阅读智能手机上的内容。可能还会有人边做饭边查看触摸屏设备上的菜谱说明。但是，如果行人缺乏视觉注意力，就会导致事故发生（第一种情况）；如果厨师手指肮脏黏腻，就会无法下滑屏幕查看剩余的菜谱（第二种情况）。这将如何是好？TTS 就是避免这些不便的自然解决方案。

可以看出，TTS 应用程序具有改善我们日常生活方方面面的潜力。

本章涵盖以下主题：

- TTS 领域概述
- 关于 TTS 的最新深度学习方法
- 分步实现 Tacotron（一个端到端深度学习模型）

## 12.1 TTS 领域概述

本节将提供有关 TTS 算法的一般信息。我们并不是要彻底解决该领域的不同问题，这将是一项非常复杂的任务，需要具有关于语言或信号处理等的跨领域知识。

我们将沿着以下高级问题：是什么使 TTS 系统变得是好或是坏？我们该如何评估？有哪些传统技术，且为什么该领域需要转向深度学习？我们还将通过提供一些有关频谱图（spectrogram）的基本信息来为下一部分做准备。

### 12.1.1 自然性与可懂性

在传统意义上，TTS 系统的质量是通过两个标准评估的：**自然性**和**可懂性**。这是由于人们不仅对音频内容很敏感，对音频内容的传送方式也很敏感。基本上，我们需要一个能以类似于人的方式产生清晰音频内容的 TTS 系统。更准确地说，可懂性是指音频质量或清晰度，而自然性是指以适当的发音、时机和情感范围传达信息。

用户可以通过具有高度可懂性的系统轻松区分不同的单词。与之相反，当可懂性较低时，某些单词可能会与其他单词混淆、难以识别，并且单词之间的分隔可能不清楚。在大多数情况下，**可懂性**是两者中更重要的参数。这是因为无论听起来自然与否，向用户传达清晰明确的消息通常是优先事项。如果用户无法理解系统生成的音频，就说明它是失败的。因此，在尝试优化所生成语音的自然性之前，必须具有最低限度的清晰度。

一方面，当 TTS 算法具有高度的自然性时，它产生的内容将非常平滑，用户此时会感觉是另一个人在说话，而无法察觉语音是人为创造的。另一方面，不连续、单调且无生气的语调是不自然语音的典型特征。



请注意，这些都是相对主观的标准，而非客观指标。实际上，由于该问题的性质，TTS 系统只能由人来评估。

### 12.1.2 TTS系统表现的评估方式

声音质量的一种主观度量指标是**平均意见得分**（mean opinion score，MOS），它是评估 TTS 算法性能的最常用测试之一。该测试通常会要求几名指定语言母语者给出自然性得分，从 1（质量差）到 5（质量优秀）。这些分数的平均值就是 MOS。专业人士录制的音频样本的 MOS 通常约为 4.55，如论文“WaveNet: A Generative Model for Raw Audio”所示，我们将在之后对其做出展示。

但是这种 TTS 基准测试算法并不完全令人满意：它不允许对不同论文中提出的不同算法进行严格的比较。实际上，算法 A 与算法 B 不一定具有相同的收听者。由于不同的人可能或多或

少地具有不同的自然声音标准，如果算法 A 的 MOS 得分为 4.2 而算法 B 的 MOS 得分为 4.1，这并不意味着算法 A 一定优于算法 B（除非它们是在同一研究中由同一组个体进行评估的）。此外，由于样本规模以及听众样本人群难以标准化，也可能有区别产生。

### 12.1.3 传统技术——级联模型和参数模型

在 TTS 任务中的深度学习技术兴起之前，人们一直在使用级联模型或参数模型。

要创建级联模型，需要录制高质量的音频内容、将其分成小块，然后重新组合以形成新的语音。对于参数模型，则必须使用信号处理技术来创建特征，而这需要一些其他领域的知识。

级联模型易于理解，但缺乏自然性，还需要庞大的数据集（该数据集应尽可能多地考虑人为生成的音频单元）。因此，该模型的开发通常需要很长。

通常来说，参数模型的性能比级联模型要差。它们生成的语音可能缺乏可懂性，而且听起来不够自然。这是因为其特征的生成过程基于人类对语音工作的认知，但我们对语音建模的方式可能是有偏见和有限制的。然而深度学习方法几乎不会受到成见的影响，且其模型可以学习到数据固有的特征。这就是深度学习的潜力所在。

### 12.1.4 关于频谱图和梅尔标度的一些提醒

正如我们将在下面看到的那样，最新的 TTS 系统（基于深度学习或其他方式）使用了一些来源于信号处理领域的技巧。例如，与直接预测波形相比，我们通常更喜欢生成频谱图非而非信号波形，最后应用转换算法。这样做可以更快地获得更好的结果。本节是对频谱图的快速回顾，将帮助你理解本章稍后将介绍的许多思想。

从本质上看，频谱图是表示音频信号强度的一种方式。它可以在二维图上显示，其中 $x$ 轴代表时间，$y$ 轴代表信号的频率。如果要加上第三个维度，则由热图表示以指示特定时间每个频率的重要性。通常冷色用于较小的幅值，而暖色则用于较大的幅值。

要计算给定数字信号的频谱图，首先需要使用**短时傅里叶变换**（short-time Fourier transform，STFT）。我们可以通过计算信号中连续帧的傅里叶变换来获得 STFT：

$$\mathrm{STFT}(m,w)=\sum_{n=-\infty}^{\infty}x[n]w[n-m]\mathrm{e}^{-jwn}, j^2=-1$$

其中，$w$ 是选择的滑动窗口。

接下来就可以获得频谱图了：

$$\mathrm{SPECTRO}(m,w)=|\mathrm{STFT}(m,w)|^2$$

实际通常使用 STFT 的非平方幅值（即绝对幅值）。

为了开发一种更接近人类感知声音的表示方式，我们有时倾向于采用梅尔标度的频谱图。对于梅尔标度，（听众）感知到的连续梅尔频率之间是等距的。它的定义是基于主观实验建立的，人们在实验中聆听不同频率的声音，然后根据自己对音高的感知程度来估计这些声音之间的距离。不同的实验会定义出不同的转换公式。最受欢迎（也是我们将要使用）的一种是：

$$\text{Mel}(f) = 2595 \times \log 10(1 + \frac{f}{700})$$

对于喜欢可视化函数的用户，我们为此公式画出了图形（如图 12-1 所示），其中 $x$ 轴表示频率，$y$ 轴表示梅尔频率。

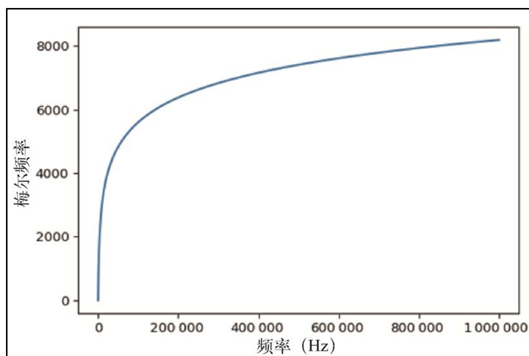



图 12-1

基本上，频谱图的频率分量通过上述公式转换为梅尔分量。此转换与**合并**操作一起完成。在合并操作中，将三角形滤波器组（$k$ 个滤波器根据梅尔标度间隔开）应用于频谱图，以便提取有限数量的梅尔频带。

图 12-2 是该滤波器组的示例，其中三角形滤波器共有 $k = 20$ 个。

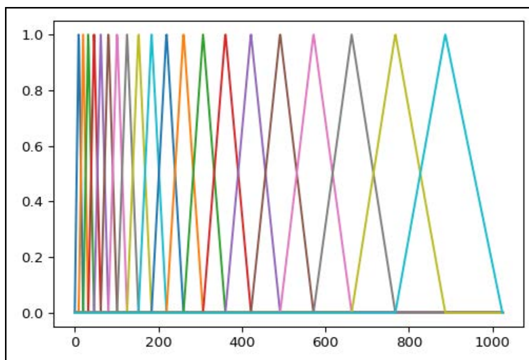



图 12-2

此外，我们更喜欢对频谱图和梅尔频谱图使用分贝标度，因为声音的幅值也被人类以对数方式感知：

$$\mathrm{SPECTRO}_{db}(m,w) = 20 \times \log 10(\mathrm{SPECTRO}(m,w))$$

我们可以使用如图 12-3 所示的音频信号（采样率为 22 050 Hz）来说明这些概念。

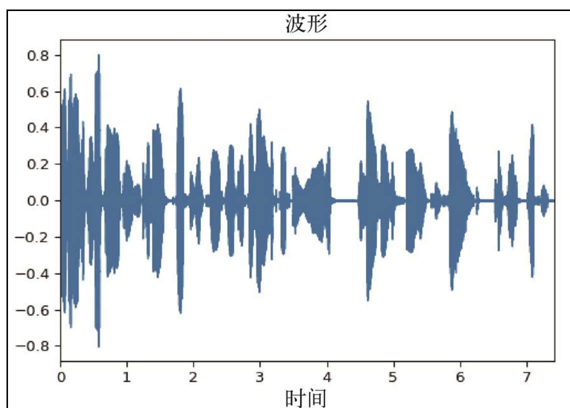



图　12-3

当查看其频谱图（以 2048 点计算出的 STFT 的大小）时，由于其对数性质，我们几乎无法区分幅值最大的部分。这就是使用分贝标度的原因，如图 12-4 所示。

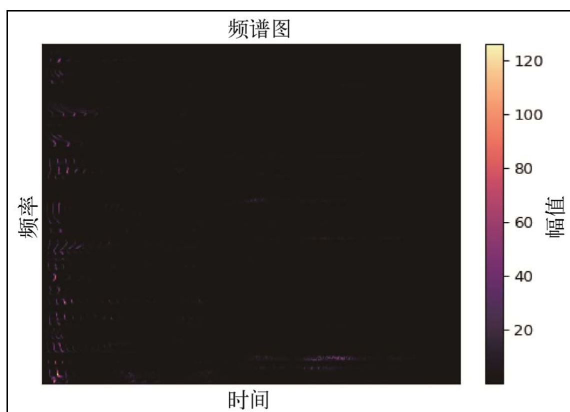



图　12-4

当使用分贝标度时，可视化结果将会更整洁，从而可以很容易地看出哪些频率随时间变化幅值最大，如图 12-5 所示。

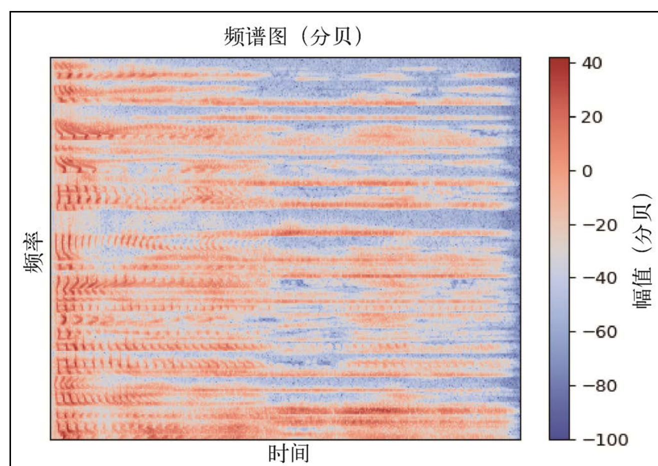



图 12-5

将梅尔标度应用于频率，再加上 80-波段滤波器组，我们可以获得更简明的频谱图表示（如图 12-6 所示），其优点是减少了频谱图矩阵的大小并减少了任何后续处理步骤中的操作次数。

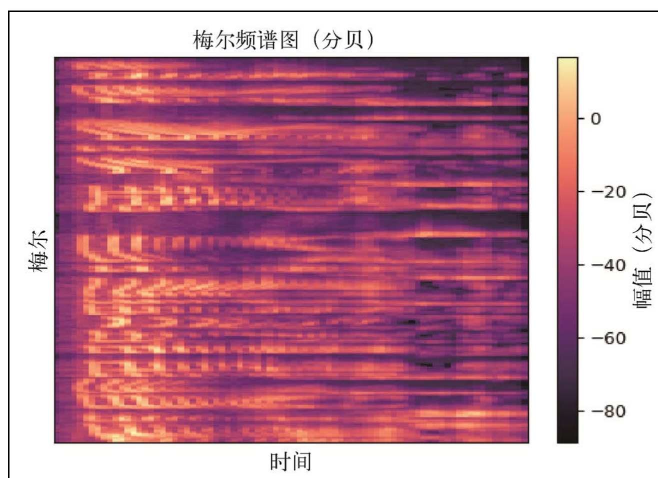



图 12-6

我们现在已整装待发，可以处理接下来的小节了。

## 12.2 深度学习中的 TTS

在过去的几年中，TTS 领域因一些基于深度学习的突破而深受影响。在这里，我们将介绍其中两个突破：WaveNet（可能是最出名的一个）和 Tacotron（具有端到端方法的优势）。

### 12.2.1　WaveNet简介

提出 WaveNet 的论文于 2016 年发表，显示出的结果优于传统的 TTS 方法。WaveNet 从根本上来说是音频生成模型，以音频样本序列作为输入，并预测最有可能的后续音频样本。通过添加额外输入，WaveNet 能完成更多任务。例如，如果在训练过程中额外提供了语音记录，WaveNet 可以将其转换为 TTS 系统。

WaveNet 使用许多有趣的思想来训练深层神经网络，主要概念涉及扩展因果卷积（请查阅其论文以了解更多）。

在论文中，该模型解决了 TTS 和其他一些任务。该模型不是直接以原始文本提供的，而是具有需要额外领域知识的工程语言特征。因此，WaveNet 并非端到端的 TTS 模型。此外，该架构非常复杂，需要大量的调整以及庞大的计算能力，才能在相当长的时间内获得令人满意的结果。

在北美英语和中文普通话数据集上对 WaveNet 展开评估，得到 MOS 并将其与级联（基于 HMM）和参数（基于 LSTM）系统进行比较得出如表 12-1 所示结果。

表　12-1

| | 北美英语 MOS | 中文普通话 MOS |
|---|---|---|
| WaveNet | 4.21 ± 0.081 | 4.08 ± 0.085 |
| 参数系统 | 3.67 ± 0.098 | 3.79 ± 0.084 |
| 级联系统 | 3.86 ± 0.137 | 3.47 ± 0.108 |

由于我们对 TTS 的端到端模型更感兴趣，因此将重点关注一个备受欢迎的模型——Tacotron。

### 12.2.2　Tacotron

Tacotron 属于首批端到端的 TTS 深度学习模型。它基本上是一个复杂的编码器–解码器模型，使用 attention 机制在文本和音频之间进行对齐。解码器产生音频的频谱图，然后通过名为 **Griffin-Lim 算法**的技术将其转换为相应的波形。

图 12-7 是其整体架构表示图。我们将在之后的段落进行详细展示。

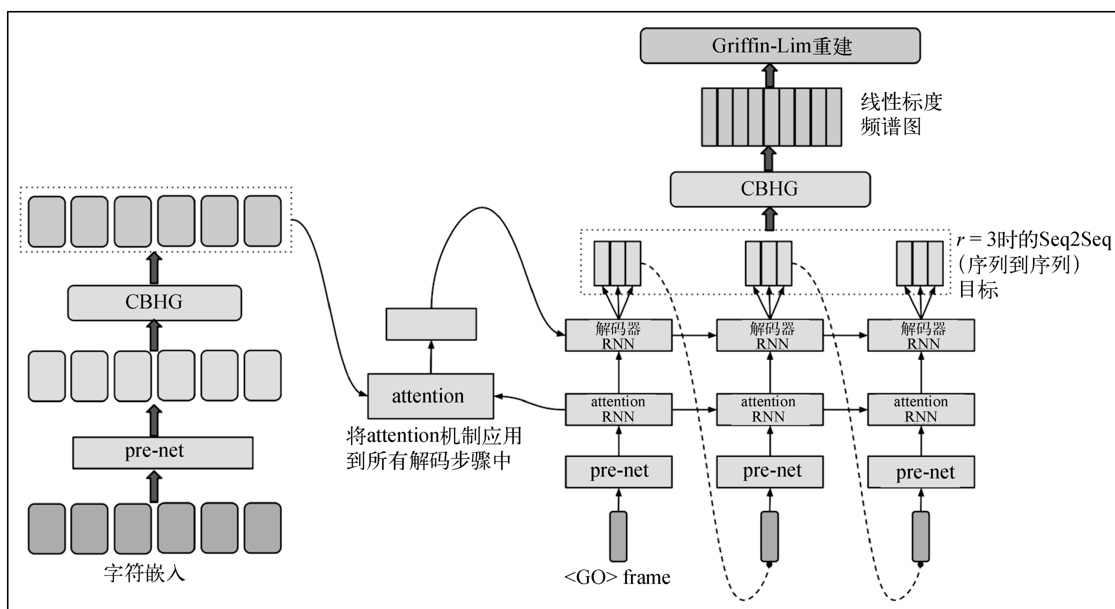



图　12-7

### 1. 编码器

编码器将一系列字符作为输入（每个字符都由独热向量表示），然后使用嵌入器将输入投射到连续的空间中。请记住，由于独热编码的高维性和稀疏性，如果不与利用该特性的技术一起使用，它可能会导致计算效率低下。嵌入器可显著减小表示空间的大小。此外，使用嵌入器可以让我们了解词汇表中不同字符之间的关系。

在嵌入层之后是一个 pre-net，它是一组非线性变换。基本上，它由两个连续的全连层组成，具有**线性整流单元**（ReLU）激活和随机失活（dropout）。dropout 是一种正则化技术，在训练过程中会忽略一些随机选择的单位（或神经元）以避免过拟合。实际在训练过程中，某些神经元会发展出相互依赖的关系，从而导致过拟合。当带有 dropout 后，神经网络倾向于学习更为稳健的特征。

第二层全连接层的单元数比第一层少一半，这有助于收敛并提高泛化的**瓶颈**层。

Tacotron 团队在 pre-net 顶部使用称作 CBHG 的模块， 该模块的名称来自构建它的基本模块：一维**卷积库**（convolution bank，CB），然后是**高速公路网络**（highway network，H）和**双向 GRU**（G）。

我们使用 $K$ 层一维卷积滤波器，形成卷积库。索引层 $K$ 包含宽度为 $k$（$k = 1, 2, \cdots, K$）的 $C_k$ 个过滤器。有了这种结构，我们应该能够为一元语法和二元语法等建模。

12

最大池化在卷积库之后立即使用。这是一种降采样技术，通常在卷积神经网络（CNN）中使用，能使得学习到的特征局部不变。在这里将其步长设置为 1 以保持时间分辨率。

论文“Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”中的技术提高了神经网络的性能和稳定性，因此在 CBHG 中的所有卷积层都有所使用。该方法修改层的输入，以使激活给出的输出的平均值为 0，标准偏差为 1。众所周知，该做法可以帮助网络更快地收敛，在深度网络中得到更高的学习率（因此朝着一个好的最小值迈出更大的步伐），从而减轻网络对权重初始化的敏感性，并通过提供一些噪声来增加额外的正则化。

在最大池化层之后，我们使用了两个补充的一维卷积层（分别具有 ReLU 和线性激活）。残差连接将初始输入与第二个卷积层的输出绑定在一起。深度网络允许捕获数据中的更多复杂性，并且在给定任务上具有比浅层网络更好的性能。但是总的来说，梯度会随着深层网络消失，而残差连接可更好地传播梯度。

因此，以上架构将能极大提升深度模型的训练效果，如图 12-8 所示。

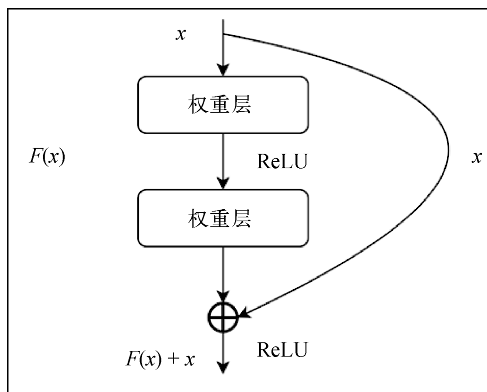



图　12-8

CBHG 的下一个块是高速公路网络，它也具有类似的作用。

为了最终确定 CBHG 模块以及编码器，我们使用双向 GRU 从向前和向后上下文中学习序列中的长期依赖关系。attention 层将使用编码器输出，如图 12-9 所示。

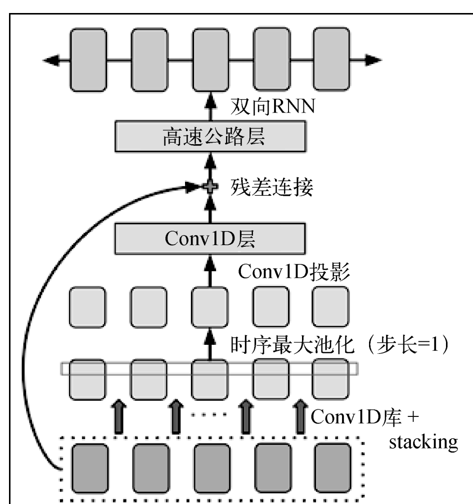



图　12-9

### 2. 基于 attention 的解码器

此处，解码器在编码器的输出上使用 attention 机制，以生成梅尔频谱图帧。

对于每个小批量数据，解码器都被提供一个 GO 帧，该帧仅包含 0 作为输入。接着，对于每个时间步，网络将先前预测的梅尔频谱图帧用作输入，馈送到与解码器中相同架构的 pre-net。

pre-net 之后是单层 GRU，其输出将与编码器的输出连接在一起，从而通过 attention 机制生成上下文向量。随后，GRU 输出也将与上下文向量连接，以生成解码器 RNN 块的输入。

正如 Wu 等人的论文“Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”所提到的，解码器 RNN 是使用垂直残差连接的两层残差 GRU。在该论文中，他们使用了一种更为复杂的残差连接。实际上，我们不仅将最后一层的输出添加到初始输入中，还在每一层上将其输出添加到其输入中，并将添加的结果用作下一层的输入，如图 12-10 所示。

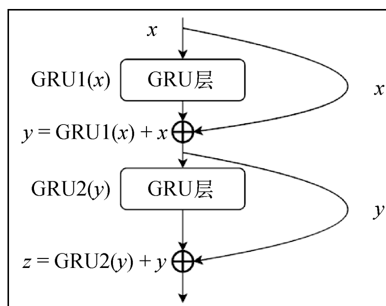



图　12-10

解码器 RNN 模块生成 $r$ 个梅尔频谱图帧。在下一个时间步中，pre-net 仅使用最后一个。之所以选择生成 $r$ 帧，而非每个时间步一个，是因为编码器输入中的一个字符通常对应于多帧。因此，通过输出一帧，我们迫使模型在多个时间步长上使用相同的输入元素，从而减慢了训练期间的 attention。论文中提到了 $r = 2$ 和 $r = 3$ 的值，其中 $r$ 的增加也会减少模型的大小和推理时间。

### 3. 基于 Griffin-Lim 的后处理模块

后处理模块的目的是将预测到的梅尔频谱图帧转换为相应的波形。

在预测得到的梅尔频谱图帧的顶部，我们使用了 CBHG 模块来提取后向和前向特征（这要归功于最后的双向 GRU）并纠正预测帧中的错误，据此可以预测得到原始频谱图。

虽然频谱图是表示语音的好方法，但它缺少有关相位的信息。幸运的是，我们有信号处理算法（例如 Griffin-Lim）可以通过估计频谱图的相位来推理可能的语音波形。 该方法通过迭代式查找以得到 STFT 幅度最接近生成频谱图的波形。

### 4. 架构细节

我们将 Adam 优化器配以特定的学习率。实际上，学习率在初始值 0.001 之后，在经过 50 万、100 万和 200 万个全局步长后将分别降低到 0.0005、0.0003 和 0.0001。一个步骤就是一个梯度更新，不应将其与周期相混淆，后者是整个训练数据集上梯度更新的完整过程。

我们将对于预测梅尔频谱图的编码器–解码器和预测频谱图的后处理块应用 L1 损失。

### 5. 局限性

Tacotron 的主要贡献无疑是提供了可理解且具有自然性的、基于深度学习的端到端 TTS 系统。实际上，Tacotron 的自然性是通过 MOS 评估的。我们将其与最先进的参数系统和最先进的级联系统（与 WaveNet 论文中的相同）进行比较（如表 12-2 所示）。虽然不如级联系统自然，但它仍击败了参数系统。请注意，MOS 是在北美英语数据集上评估的。

表 12-2

| | 北美英语平均意见得分 |
|---|---|
| Tacotron | 3.82 ± 0.085 |
| 参数系统 | 3.69 ± 0.109 |
| 级联系统 | 4.09 ± 0.119 |

但要记住的是，Tacotron 具有许多简单的设计选择。例如，Griffin-Lim 重建算法既轻巧又简单，但会造成干扰，对自然性产生负面影响。使用更强大的技术来替换这一部分的管道可能会进一步增加 MOS。我们还可以对许多其他部分进行调整和改进：模型的超参数、attention 机制、学习率计划表、损失函数，等等。

现在我们已充分了解 Tacotron 的运行机制，可以去实现代码了。

## 12.3 利用 Keras 的 Tacotron 实现

本节将通过在 TensorFlow 上使用 Keras 展示 Tacotron 的实现。Keras 与原始 TensorFlow 相比的优势在于更快的原型制作和高度模块化。但就灵活性而言，TensorFlow 还是比 Keras 更好一些，尽管更难学习。目前，TensorFlow 还提供了更多内置功能（例如 attention 机制），其中一些将会在此处被重新实现。

我们将使用 Keras 2.1.5，配有 TensorFlow 1.6.0 作为其后端。

下面介绍代码库的组织方式。

- /data 文件夹包含原始数据集，并将通过几个处理步骤进行增强。
- /model 文件夹包含：
  - building_blocks.py，它定义了 Tacotron 模型的所有基本单元；
  - tacotron_model.py，它将创建 Tacotron 模型。
- /processing 文件夹包含：
  - proc_audio.py，它提供音频处理功能，使我们能够将波形转换为频谱图；
  - proc_text.py，它允许将原始转录文本转换为更适合深度学习的格式。
- /results 文件夹将包含训练好的模型，并记录它们的损失：
  - 1_create_audio_dataset.py，它生成训练和测试音频数据（模型目标）；
  - 2_create_text_dataset.py，它生成训练和测试文本数据（模型输入）；
  - 3_train.py，它使用训练数据训练模型，并保存模型及损失历史记录；
  - 4_test.py，它在测试数据集的选定项目上测试最后训练的模型；
  - constants.py，它包含所有必需的常量。

表 12-3 展示了此项目使用的最为重要的 Python 模块。我们没有提到不同的依赖项，因为当 `pip install` 被触发时会自动安装它们。

表 12-3

| 模块名称 | 描　述 |
|---|---|
| pandas | 用于数据读入、处理和分析 |
| NumPy | 提供用于科学计算的数据结构和方法 |
| Scikit-learn | 包含许多关于机器学习的处理方法 |
| TensorFlow | 本章中用作 Keras 后端的深度学习框架 |
| Keras | 用于设计神经网络的简单、模块化的高层 API |
| Librosa | 提供音频处理函数 |
| tqdm | 允许显示进度条以跟踪 `for` 循环的演变 |
| Matplotlib | 可用于可视化预估和实际频谱图、波形 |

## 12.3.1　数据集

我们将使用 LJ 语音数据集来执行此任务。它包含 13 100 个.wav 录音及其相应的转录文本。转录文本有其原始格式和规范化格式。在转录文本的规范化版本中，数是用完整的单词写成的。

录音采用相同的声音制作。音频内容的总长度约为 24 小时，采样时间可持续 1 秒到 7 秒。该数据集属于公共领域，并无使用限制。



注意，该数据集下载并解压缩后将占据磁盘 3.8 GB 的空间。

数据集文件夹中有一个名为 metadata.csv 的 CSV 文件、一个 README 文件和一个包含.wav 音频文件的文件夹/wavs。metadata.csv 由三列组成，共 13 100 行。第一列给出了相应的.wav 文件的名称，其他两列分别给出了原始和标准化的转录文本，如图 12-11 所示。

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | LJ001-0001 | Printing, in the only sense with which we are ... | Printing, in the only sense with which we are ... |
| 1 | LJ001-0002 | in being comparatively modern. | in being comparatively modern. |
| 2 | LJ001-0003 | For although the Chinese took impressions from... | For although the Chinese took impressions from... |

图　12-11

下载完毕后，应将包含数据的 ZIP 文件解压缩到/data 目录下。

## 12.3.2　数据准备

为了训练 Tacotron，我们需要在此数据集上应用几个预处理步骤。首先必须在 metadata.csv 中准备规范化的文本数据，使其具有正确的形状以用作编码器的输入。同样，我们应该提取分别由解码器和 CBHG 后处理模块输出的梅尔频谱图和幅度频谱图。

可以使用 pandas `read_csv` 加载数据。我们需要考虑以下事实：CSV 文件不包含任何标题，使用竖线字符分隔各列，并且包含未结束的引号（转录下来的并不总是完整的句子）：

```
metadata = pd.read_csv('data/LJSpeech-1.1/metadata.csv',
                       dtype='object', quoting=3, sep='|',
                       header=None)
```

我们决定将 90%的数据（117.9 亿项）用于训练，而将其余 10%的数据（1.31 亿项）用于测试。该比例可以任意选择，我们将定义一个变量 `TRAIN_SET_RATIO`，你可以用其进行调整：

```
TRAIN_SET_RATIO = 0.9
```

### 1. 文本数据准备

与典型 NLP 任务情况一样，所有字符串都将被转换为小写。由于模型将考虑字符序列（而非单词序列），我们将训练词汇表作为数据集使用的唯一字符集。我们将添加一个与填充相对应的字符 P，因为需要定义一个固定的输入长度 NB_CHARS_MAX，并用填充字符补满长度不足的字符串。

```
list_of_existing_chars = list(set(texts.str.cat(sep=' ')))
vocabulary = ''.join(list_of_existing_chars)
vocabulary += 'P' # 添加填充字符
```

任意字符都将用一个整数进行标识：

```
# 在词汇和 ID 之间建立连接
vocabulary_id = {}
i = 0
for char in list(vocabulary):
    vocabulary_id[char] = i
    i += 1
```

现在，我们已准备好转换文本数据。定义一个函数 transform_text_for_ml，它接受字符串列表 list_of_strings（其中每个字符串都是转录文本）、字典 vocabulary_ids（用以将词汇表中的字符映射为整数）以及每个转录文本的最大字符数 max_length。此函数将转录文本转换为字符列表（按照其出现顺序），并根据需要添加尽可能多的填充字符（以便使句子有 max_length 个字符）：

```
def transform_text_for_ml(list_of_strings, vocabulary_ids, max_length):
    transformed_data = []

    for string in tqdm(list_of_strings):
        list_of_char = list(string)
        list_of_char_id = [vocabulary_ids[char] for char in list_of_char]

        nb_char = len(list_of_char_id)

        # 填充达到固定输入长度
        if nb_char < max_length:
            for i in range(max_length - nb_char):
                list_of_char_id.append(vocabulary_ids['P'])
        transformed_data.append(list_of_char_id)

    ml_input_training = np.array(transformed_data)

    return ml_input_training
```

我们现在可以应用这个处理函数了：

```
text_input_ml = transform_text_for_ml(texts.values,
                                      vocabulary_id,
                                      NB_CHARS_MAX)
```

Python 脚本 2_create_text_dataset.py 将加载文本数据并对其进行处理（如前所示），将结果分为训练集和测试集并都转储为 pickle 文件。同时，脚本还保存了用于映射字符及其关联整数的词汇表：

```
# 划分为训练集和测试集
len_train = int(TRAIN_SET_RATIO * len(metadata))
text_input_ml_training = text_input_ml[:len_train]
text_input_ml_testing = text_input_ml[len_train:]

# 保存数据
joblib.dump(text_input_ml_training, 'data/text_input_ml_training.pkl')
joblib.dump(text_input_ml_testing, 'data/text_input_ml_testing.pkl')

joblib.dump(vocabulary_id, 'data/vocabulary.pkl')
```

### 2. 语音数据准备

本节将使用信号处理领域的许多术语，我们将解释其中的一部分，其余部分则不作要求。本书本身就是关于深度学习的，我们鼓励好奇的读者对尚未完全解释的信号处理概念进行深入研究。

我们将使用如表 12-4 所示的（从论文中获得的）参数进行.wav 文件的光谱分析和处理。

表　12-4

| 变　量　名 | 描　　述 | 值 |
|---|---|---|
| N_FFT | 傅里叶变换点数 | 1024 |
| PREEMPHASIS | 预加重技术的参数，更加重视信号中的高频分量 | 0.97 |
| SAMPLING_RATE | 采样率 | 16 000 |
| WINDOW_TYPE | 用于计算傅里叶变换的窗口类型 | ‘hann’ |
| FRAME_LENGTH | 窗口长度 | 50 ms |
| FRAME_SHIFT | 时移 | 12.5 ms |
| N_mels | 梅尔波段数 | 80 |
| r | 衰减系数 | 5 |



尽管音频信号的原始采样率为 22.05 kHz，我们 数据处理时还是决定使用 16 kHz，这样可以减少计算操作的数量。此外，论文中建议的傅里叶变换点数为 2048，而我们在这里使用 1024 点，以进一步减轻任务负担。

Tacotron 模型优化了两个目标函数：一个用于从解码器 RNN 输出的梅尔频谱图，另一个用于 CBHG 后处理应用梅尔频谱图所给出的频谱图输出。因此我们需要准备这两种类型的输出。

从.wav 文件的路径下返回信号的频谱图和梅尔频谱图：

```
def get_spectros(filepath, preemphasis, n_fft,
                 hop_length, win_length,
                 sampling_rate, n_mel,
                 ref_db, max_db):
    waveform, sampling_rate = librosa.load(filepath,
                                           sr=sampling_rate)

    waveform, _ = librosa.effects.trim(waveform)

    # 使用预加重滤除较低频率
    waveform = np.append(waveform[0],
                         waveform[1:] - preemphasis * waveform[:-1])

    # 计算 stft
    stft_matrix = librosa.stft(y=waveform,
                               n_fft=n_fft,
                               hop_length=hop_length,
                               win_length=win_length)

    # 计算幅值和梅尔频谱图
    spectro = np.abs(stft_matrix)

    mel_transform_matrix = librosa.filters.mel(sampling_rate,
                                               n_fft,
                                               n_mel,
                                               htk=True)
    mel_spectro = np.dot(mel_transform_matrix,
                         spectro)

    # 使用分贝标度
    mel_spectro = 20 * np.log10(np.maximum(1e-5, mel_spectro))
    spectro = 20 * np.log10(np.maximum(1e-5, spectro))

    # 标准化频谱图
    mel_spectro = np.clip((mel_spectro - ref_db + max_db) / max_db, 1e-8,
1)
    spectro = np.clip((spectro - ref_db + max_db) / max_db, 1e-8, 1)

    # 将频谱图转置，使得时间为第一维，
    # 频率为第二维
    mel_spectro = mel_spectro.T.astype(np.float32)
    spectro = spectro.T.astype(np.float32)

    return mel_spectro, spectro
```

如果频谱图的总长度不是 r 的倍数，则需要填充频谱图的时间维度，从而使其满足要求：

```
def get_padded_spectros(filepath):
    filename = os.path.basename(filepath)
    mel_spectro, spectro = get_spectros(filepath)
    t = mel_spectro.shape[0]
    nb_paddings = r - (t % r) if t % r != 0 else 0 # for reduction
    mel_spectro = np.pad(mel_spectro,
                         [[0, nb_paddings], [0, 0]],
                         mode="constant")
```

```
    spectro = np.pad(spectro,
                     [[0, nb_paddings], [0, 0]],
                     mode="constant")
    return filename, mel_spectro.reshape((-1, N_mel * r)), spectro
```

通过 1_create_audio_dataset.py 脚本将 `get_padded_spectros` 应用于数据集的所有.wav 文件。它会将所有频谱图和梅尔频谱图生成为数组以及解码器的输入，并将这三个数组分成训练集和测试集，就像对处理后的文本数据所做的那样。

请注意，运行脚本可能需要很长时间（最多几个小时），这就是还要对所得数据进行 pickle 序列化的原因。这样就不需要每次试图训练模型时都重新处理文件了。

### 12.3.3　架构实现

当数据准备就绪后，我们就可以构建模型了。我们将从实现网络构建模块开始，然后将它们结合起来。整个过程都是在/model 中完成的。

#### 1. pre-net

如论文中所述，我们将实现 pre-net 块，编码器和解码器中都会用到它。Keras 的简单性和模块化使得此部分非常简单：

```
def get_pre_net(input_data):
    prenet=Dense(256)(input_data)
    prenet=Activation('relu')(prenet)
    prenet=Dropout(0.5)(prenet)
    prenet=Dense(128)(prenet)
    prenet=Activation('relu')(prenet)
    prenet=Dropout(0.5)(prenet)
    return prenet
```

#### 2. 编码器和 CBHG 后处理

为了准备 CBHG 模块的代码，我们首先实现它的两个主要基本单元—— 一维卷积库和高速公路网络：

```
def get_conv1dbank(K_, input_data):
    conv=Conv1D(filters=128, kernel_size=1,
                strides=1,padding='same')(input_data)
    conv=BatchNormalization()(conv)
    conv=Activation('relu')(conv)

    for k_ in range(2,K_+1):
        conv=Conv1D(filters=128, kernel_size=k_,
                    strides=1,padding='same')(conv)
        conv=BatchNormalization()(conv)
        conv=Activation('relu')(conv)

  return conv
```

对于 Keras 2，其开发团队决定删除高速公路网络，可能是因为它们很少被用到并且容易实现。因为我们使用的是 Keras 2，所以需要显式地编写自己的高速公路层。我们根据先前引用的高速路网络论文来定义 get_highway_output 函数：给定输入张量 highway_input 时，返回由 nb_layers 层定义的高速公路网络的输出、激活函数 activation 和 initial_bias（根据高速公路网络的论文，initial_bias 通常为-1 或-3）：

```
def get_highway_output(highway_input, nb_layers, activation="relu",
bias=-3):
    dim = K.int_shape(highway_input)[-1] # 维度必须相同
    initial_bias = k_init.Constant(bias)
    for n in range(nb_layers):
        H = Dense(units=dim, bias_initializer=initial_bias)(highway_input)
        H = Activation("sigmoid")(H)
        carry_gate = Lambda(lambda x: 1.0 - x,
                            output_shape=(dim,))(H)
        transform_gate = Dense(units=dim)(highway_input)
        transform_gate = Activation(activation)(transform_gate)
        transformed = Multiply()([H, transform_gate])
        carried = Multiply()([carry_gate, highway_input])
        highway_output = Add()([transformed, carried])
    return highway_output
```

现在，我们准备实现 CBHG 模块。编码器和后处理 CBHG 的体系结构略有不同。虽然可以只编写一个带有额外输入参数的函数，我们还是决定编写两个不同的函数，以提高代码可读性：

```
def get_CBHG_encoder(input_data, K_CBHG):
    conv1dbank = get_conv1dbank(K_CBHG, input_data)
    conv1dbank = MaxPooling1D(pool_size=2, strides=1,
                              padding='same')(conv1dbank)
    conv1dbank = Conv1D(filters=128, kernel_size=3,
                        strides=1, padding='same')(conv1dbank)
    conv1dbank = BatchNormalization()(conv1dbank)
    conv1dbank = Activation('relu')(conv1dbank)
    conv1dbank = Conv1D(filters=128, kernel_size=3,
                        strides=1, padding='same')(conv1dbank)
    conv1dbank = BatchNormalization()(conv1dbank)

    residual = Add()([input_data, conv1dbank])

    highway_net = get_highway_output(residual, 4, activation='relu')

    CBHG_encoder = Bidirectional(GRU(128,

return_sequences=True))(highway_net)

    return CBHG_encoder

def get_CBHG_post_process(input_data, K_CBHG):
    conv1dbank = get_conv1dbank(K_CBHG, input_data)
    conv1dbank = MaxPooling1D(pool_size=2, strides=1,
                              padding='same')(conv1dbank)
```

```
    conv1dbank = Conv1D(filters=256, kernel_size=3,
                        strides=1, padding='same')(conv1dbank)
    conv1dbank = BatchNormalization()(conv1dbank)
    conv1dbank = Activation('relu')(conv1dbank)
    conv1dbank = Conv1D(filters=80, kernel_size=3,
                        strides=1, padding='same')(conv1dbank)
    conv1dbank = BatchNormalization()(conv1dbank)

    residual = Add()([input_data, conv1dbank])

    highway_net = get_highway_output(residual, 4, activation='relu')

    CBHG_post_proc = Bidirectional(GRU(128))(highway_net)

    return CBHG_post_proc
```

### 3. attention RNN

如前所述，attention RNN 是一个简单的单层 GRU。按照论文的定义，它包含 256 个单位。为其定义一个函数看起来是大材小用，但这可以提高代码的可读性，尤其是因为我们已经使用论文中的术语描述了该架构：

```
def get_attention_RNN():
    return GRU(256)
```

### 4. 解码器 RNN

解码器 RNN 是具有垂直残留连接的双层 GRU（如前所述）：

```
def get_decoder_RNN_output(input_data):
    rnn1 = GRU(256, return_sequences=True)(input_data)

    inp2 = Add()([input_data, rnn1])
    rnn2 = GRU(256)(inp2)
    decoder_rnn = Add()([inp2, rnn2])

    return decoder_rnn
```



请注意，在定义第一个 GRU 层时必须使用 `return_sequences = True`。这样，对于每个输入时间步长，将返回一个输出。此时给定一个序列作为输入，第一个 GRU 也将输出一个序列。如果不这样做，则第一个 GRU 对于整个输入序列仅会返回一个输出，而第二个 GRU 期望以序列作为输入。

### 5. attention 机制

网络首先将基于 CBHG 编码器的双向 GRU 层生成的输出向量与 attention RNN 的输出进行级联，获得 attention 上下文。然后，将结果向量馈入 tanh 激活的密集层，在这之后是另一个密集层，而 softmax 层允许通过与编码器输出向量的点乘积来获得给出 attention 上下文的激活权重：

```
def get_attention_context(encoder_output,attention_rnn_output):
    attention_input=Concatenate(axis=-1)([encoder_output,
                                          attention_rnn_output])
    e=Dense(10, activation = "tanh")(attention_input)
    energies=Dense(1, activation = "relu")(e)
    attention_weights=Activation('softmax')(energies)
    context=Dot(axes = 1)([attention_weights,
                           encoder_output])
    return context
```

### 6. 带有 attention 的完整架构

现在来将先前定义的函数结合起来以形成完整的 Tacotron 模型。

不过首先需要定义网络的一些特征参数：

```
NB_CHARS_MAX = 200 # 输入文本最大长度
EMBEDDING_SIZE = 256

K1 = 16 # 在编码器 CBHGH 中一维卷积块个数
K2 = 8 # 在后处理 CBHGH 中一维卷积块个数

BATCH_SIZE = 32
```



注意，模型分别由两个输入对象和两个输出对象定义。

这两个输入对象分别对应于编码器输入和解码器输入。前者应为输入文本，而后者应该是解码器在 CBHG 后处理前所预测的 $r$ 帧中最后一个梅尔频谱图帧。解码器输入的第一帧被填充为 0，如论文中所示。

输出对应于解码器 RNN 预测的梅尔标度频谱图和 CBHG 后处理模块预测的频谱图：

```
def get_tacotron_model(n_mels, r, k1, k2, nb_char_max,
                       embedding_size, mel_time_length,
                       mag_time_length, n_fft,
                       vocabulary):
    # 编码器
    input_encoder = Input(shape=(nb_char_max,))

    embedded = Embedding(input_dim=len(vocabulary),
                         output_dim=embedding_size,
                         input_length=nb_char_max)(input_encoder)
    prenet_encoding = get_pre_net(embedded)

    cbhg_encoding = get_CBHG_encoder(prenet_encoding,
                                     k1)

    # 解码器第一部分——prenet
    input_decoder = Input(shape=(None, n_mels))
    prenet_decoding = get_pre_net(input_decoder)
```

12

```
    attention_rnn_output = get_attention_RNN()(prenet_decoding)

    # attention
    attention_rnn_output_repeated = RepeatVector(
        nb_char_max)(attention_rnn_output)

    attention_context = get_attention_context(cbhg_encoding,
attention_rnn_output_repeated)

    context_shape1 = int(attention_context.shape[1])
    context_shape2 = int(attention_context.shape[2])
    attention_rnn_output_reshaped = Reshape((context_shape1,
context_shape2))(attention_rnn_output)

    # 解码器第二部分
    input_of_decoder_rnn = concatenate(
        [attention_context, attention_rnn_output_reshaped])
    input_of_decoder_rnn_projected = Dense(256)(input_of_decoder_rnn)

    output_of_decoder_rnn = get_decoder_RNN_output(
        input_of_decoder_rnn_projected)

    # mel_hat=TimeDistributed(Dense(n_mels*r))(output_of_decoder_rnn)
    mel_hat = Dense(mel_time_length * n_mels * r)(output_of_decoder_rnn)
    mel_hat_ = Reshape((mel_time_length, n_mels * r))(mel_hat)

    def slice(x):
        return x[:, :, -n_mels:]

    mel_hat_last_frame = Lambda(slice)(mel_hat_)
    post_process_output = get_CBHG_post_process(mel_hat_last_frame,
                                                k2)

    z_hat = Dense(mag_time_length * (1 + n_fft // 2))(post_process_output)
    z_hat_ = Reshape((mag_time_length, (1 + n_fft // 2)))(z_hat)

    model = Model(inputs=[input_encoder, input_decoder],
                  outputs=[mel_hat_, z_hat_])
    return model
```

然后就可以编译模型了。由于定义了两个输出对象，我们需要两个损失函数。论文中选取了两个 L1 损失，并且其权重相等，我们也决定沿用该选择。此外，在默认情况下，Adam 及其参数会被配置好以用作优化器。为了简单起见，我们决定不遵循论文中所使用的学习率计划，但是鼓励你尝试更高级的设置：

```
opt = Adam()
model.compile(optimizer=opt,
              loss=['mean_absolute_error','mean_absolute_error'])
```

### 12.3.4　训练与测试

完成上述步骤后，可以通过 3_train.py 和 4_test.py 进行训练和测试。

第一个脚本在准备好的训练集上对 Tacotron 模型训练了 `NB_EPOCHS` 个周期，并将模型存储在了/results 文件夹下。

第二个脚本允许用户将先前保存的模型应用于测试数据集的任何副本。通过变量 `item_index` 选择要预测的音频，该变量应包含所需项目的索引（在测试数据集中）。

然后，通过 Griffin-Lim 算法将估算的频谱图转换为波形。转换函数 `from_spectro_to_waveform` 是在/processing/proc_audio.py 文件中定义的。

我们强烈建议你更改代码库的默认设置，尝试更高级的方法，以提高生成波形的质量或模型的收敛速度。

## 12.4 小结

本章概述了 TTS 领域，解释了良好 TTS 系统应遵循的标准，并探索了传统 TTS 方法的冰山一角。

然后，我们提出了一种最新的、端到端的深度学习方法 Tacotron 并在本章末尾对其进行了实现，同时给出了在适应该问题的开源数据集上对该模型进行实验的指导和说明。

在 12.3 节中，我们看到了 Keras 简化看似复杂的神经网络构建过程。但是，制作原型是一回事，而扩大规模却是另一回事。实际上，即使验证概念能在计算机上顺利运行，但由于多种原因（例如吞吐量），设法使它在不同平台（网络和移动平台）上供大量用户使用也可能具有挑战性。

下一章会解决将深度学习模型交付到生产环境的问题。

# 第 13 章 部署训练好的模型

在本章中，你将学习如何将训练好的深度学习模型部署到各种平台（例如云和移动设备等）上的生产环境中。对于云部署，延迟和吞吐量很重要：它要求延迟达到最小，且同时吞吐量必须很高。模型部署后的性能在很大程度上取决于其自身和硬件。现在有多种针对 CPU 和 GPU 的优化方法。对于移动平台而言，速度和能耗都很重要。

在本章中，你将通过以下主题学习相关技术以达成部署目标：

- 通过更改模型来提高性能
- 使用 TensorFlow Serving 工具
- 部署到云服务，例如 AWS、GCP 和 Azure
- 部署到如 iPhone、Android 和 Tegra 等移动设备
- 硬件对性能的影响

## 13.1 性能提升

模型的推理时间取决于使用硬件运行模型所需的**每秒浮点运算**（floating-point operations per second，FLOPS）。FLOPS 受所涉及的模型参数数量和浮点运算次数的影响。浮点运算主要是矩阵运算，例如加法、乘法和除法。例如，卷积运算只有几个代表内核的参数，但是计算时间较长，因为该运算必须在输入矩阵上执行。对于全连接层，虽然参数多，但是运行很快。

模型的权重通常是双精度浮点值或高精度浮点值。对此类数字进行算术运算要比对量化值进行运算昂贵。下一节将说明量化权重是如何影响模型性能的。

### 13.1.1 量化权重

量化模型的权重会降低其浮点精度。尽管精度有所降低，但模型的权重仍可以有合理的准确率表现。现代硬件可以在较短的时间内执行精度较低的操作。

## 13.1.2　MobileNets

Howard 等人提出了一种使用学习范式进行更快推理的解决方案。图 13-1 是对各种模型使用移动推理的展示。通过这种技术生成的模型也可以用于云服务。

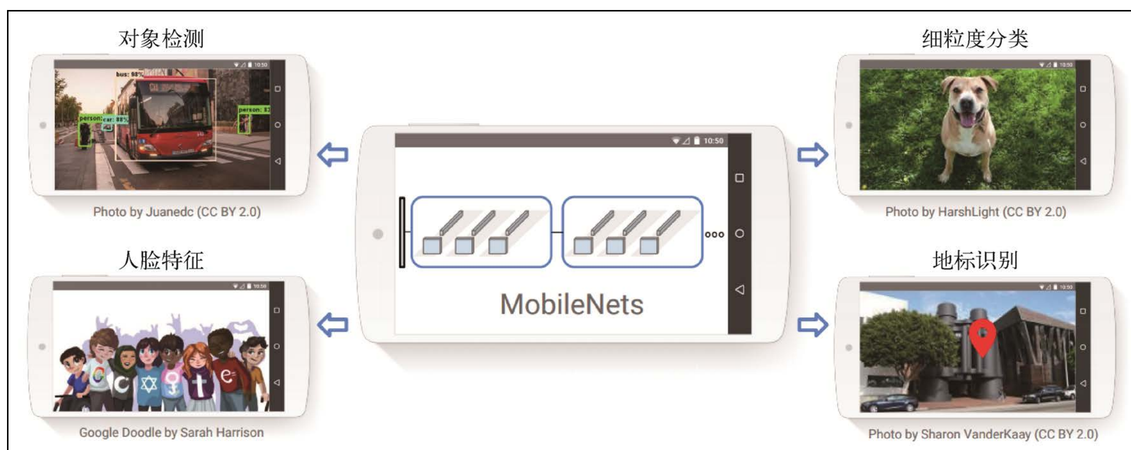



图 13-1　对各种模型运用移动推理，来自 Howard 等人

有三种应用卷积的方法，如图 13-2 所示。

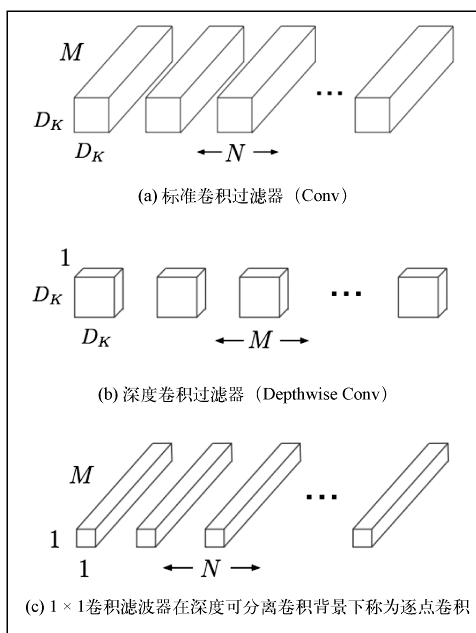



图 13-2　来自 Howard 等人

常规卷积可以用深度卷积代替，如图 13-3 所示。

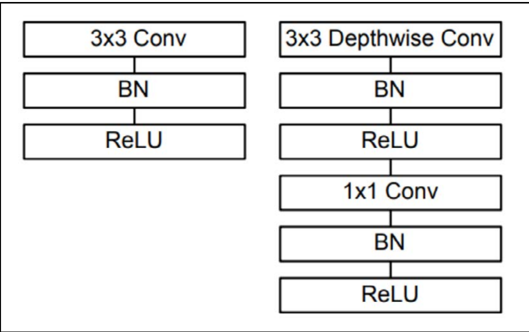



图 13-3　来自 Howard 等人

图 13-4 是表示准确率与所执行操作数的线性关系的图。

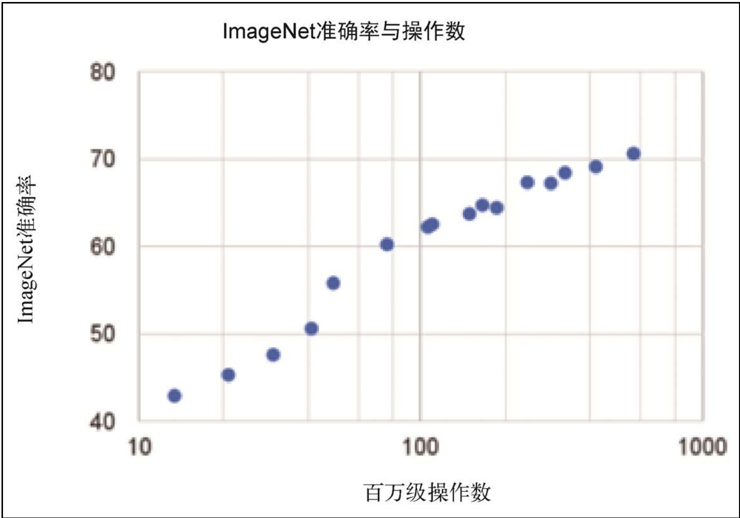



图 13-4　来自 Howard 等人

图 13-5 表示了精度对参数数量的依赖性，其中参数以对数刻度绘制。

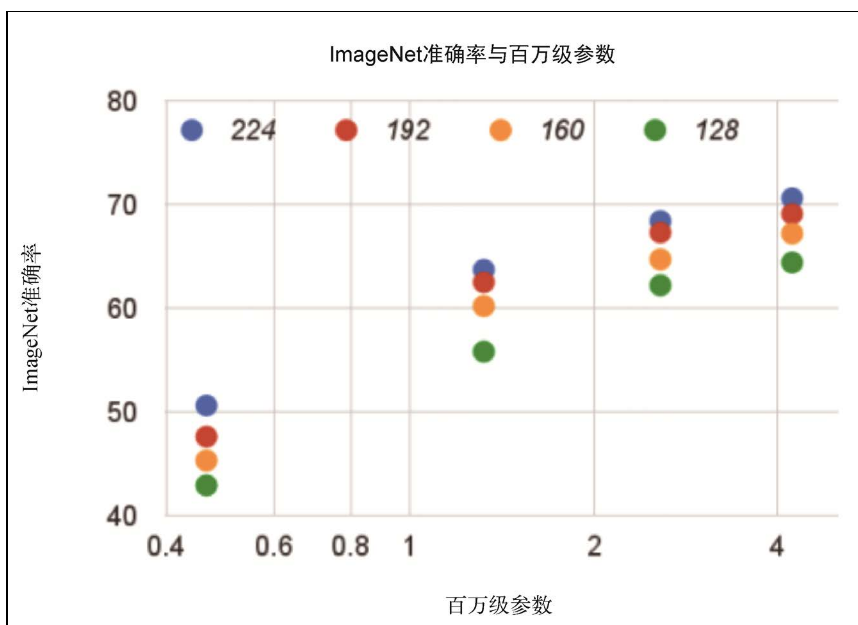



图 13-5 来自 Howard 等人

从前面的讨论中可以清楚地看出，量化提高了模型推理的性能。在下一节中，我们将了解如何使用 TensorFlow Serving 为生产中的模型提供服务。

## 13.2 TensorFlow Serving

TensorFlow Serving 是 Google 的一个项目，可将模型部署到生产环境中。TensorFlow Serving 具有以下优点：

- 延迟低、推理快；
- 并行式，提供良好的吞吐量；
- 模型版本管理，可以交换模型而不会导致生产停机。

这些优势使 TensorFlow Serving 成为部署到云的出色工具。TensorFlow 由 gRPC 服务器（Google 的远程过程调用系统）提供服务。大多数生产环境在 Ubuntu 上运行，因此安装 TensorFlow 服务的最简单方法是使用 `apt-get`，如下所示：

```
sudo apt-get install tensorflow-model-serving
```

我们也可以在其他环境下对源代码进行编译。由于使用 Docker 进行部署的普遍性，将其构建为 Ubuntu 映像更加容易。

图 13-6 是 TensorFlow Serving 的架构图。

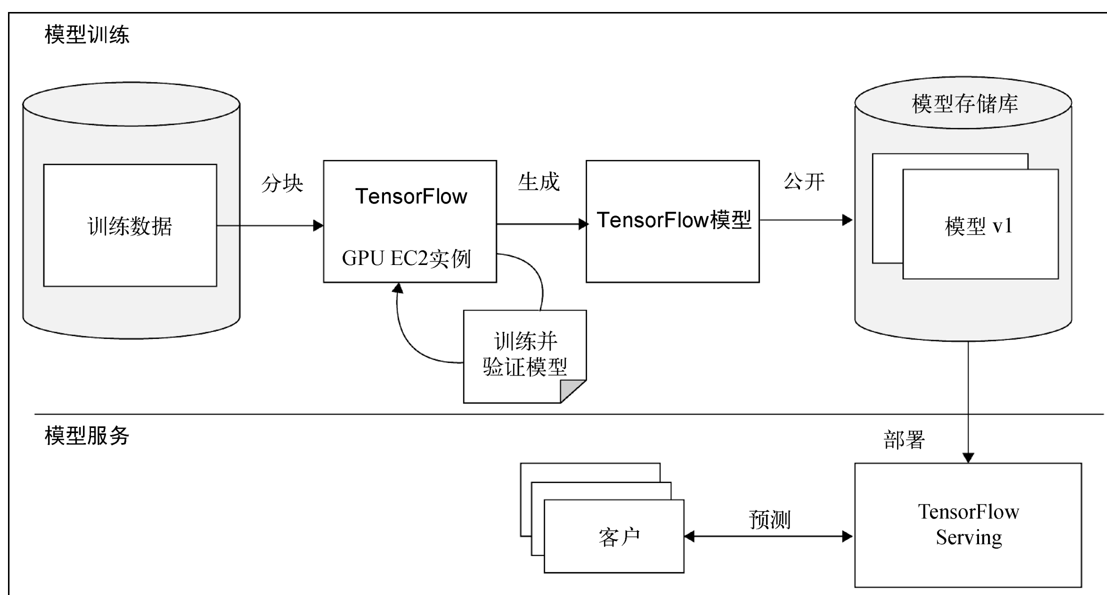



图　13-6

模型经过训练和验证之后，就可以将其推送到模型存储库。TensorFlow Serving 将开始根据版本号为模型提供服务。客户可以使用 TensorFlow Serving 客户端查询服务器。要从 Python 使用客户端，请按以下步骤安装 TensorFlow Serving API：

```
sudo pip3 install tensorflow-serving-api
```

先前的命令将安装 TensorFlow Serving 的客户端组件，然后就可以将其用于对服务器的推理调用了。

## 13.2.1　导出训练好的模型

我们可以通过使用 `tf.saved_model.builder.SavedModelBuilder` 作为协议缓冲区对象来保存训练后的模型。首先创建构建器、输入和输出张量。下面的伪代码用于说明该过程，你必须用相应的变量替换尖括号（<>）包围的相应变量：

```
bldr = tf.saved_model.builder.SavedModelBuilder(<directory_to_export>)
tensor_inputs = tf.saved_model.utils.build_tensor_info(<inputs>)
tensor_outputs = tf.saved_model.utils.build_tensor_info(<outputs>)
```

TensorFlow `utils.build_tensor_info` 实用函数可帮助我们创建必要的输入和输出协议缓冲区。接下来，我们将为推理协议缓冲区创建签名：

```
inference_signature = (
tf.saved_model.signature_def_utils.build_signature_def(inputs={'inputs':
tensor_inputs},
    outputs={'predictions': tensor_outputs},
    method_name= tf.saved_model.signature_constants.PREDICT_METHOD_NAME)
```

TensorFlow `signature_def_utils.build_signature_def` 函数可用于创建签名定义协议缓冲区。创建推理签名后，我们将使用模型构建器的 `add_meta_graph_and_variables` 函数将其导出并保存：

```
bldr.add_meta_graph_and_variables(tflow_s,[tf.saved_model.tag_constants.SER
VING],
    signature_def_map={ 'inference_results': inference_signature},
    legacy_init_op=<op_init>)
bldr.save()
```

这会将模型另存为 export 目录下的协议缓冲区。

### 13.2.2 把导出模型投入服务

导出模型后，可以通过 `tensorflow_model_server` 命令来启动或提供服务：

```
tensorflow_model_server --port=9000 --model_name=<modelname> --
model_base_path=<modelpath>
```

你可以使用相应的模型名称和路径来替换尖括号包围的部分。

## 13.3 在云上部署

有许多云供应商可用于部署训练好的模型。在本节中，我们将看到使用 Amazon Web Services（AWS）和 Google Cloud Platform（GCP）的部署步骤。

### 13.3.1 Amazon Web Services

AWS 提供了许多产品和解决方案来支持机器学习。首先按以下步骤，看看如何使用 AWS 启动一台简单的机器。

(1) 登入 AWS 后，你将看到如图 13-7 所示屏幕。

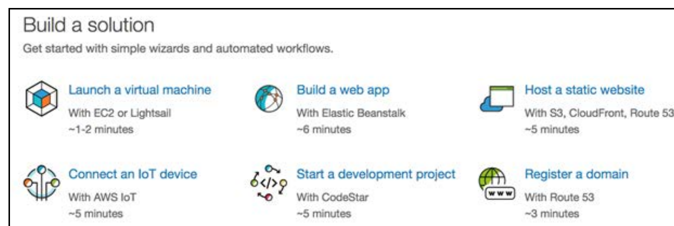



图 13-7

(2) 选择 Launch a virtual machine（运行虚拟机）选项。接下来你将看到如图 13-8 所示画面。

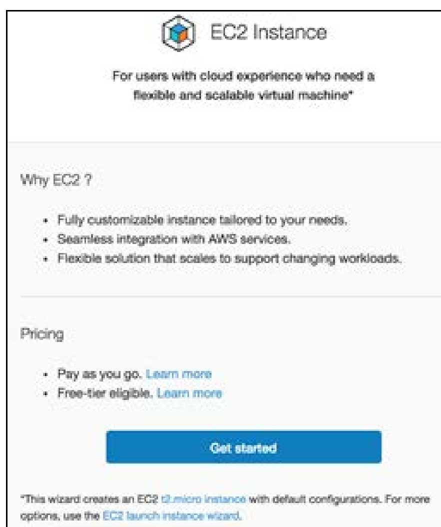



图 13-8　AWS 启动画面

(3) 单击 Get started（开始）进入下一画面，如图 13-9 所示。

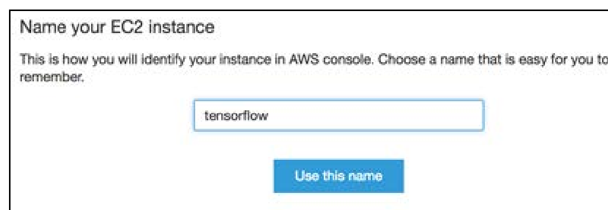



图 13-9　输入你的实例识别名

(4) 输入名称（可能是你自己的名字）进入下一画面，如图 13-10 所示。

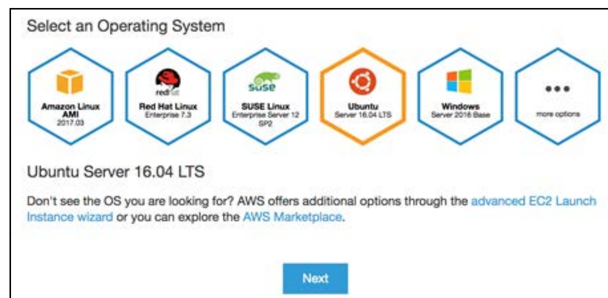



图 13-10　选择操作系统

(5) 选择用于部署的操作系统。Ubuntu 是个不错的选择，如图 13-11 所示。

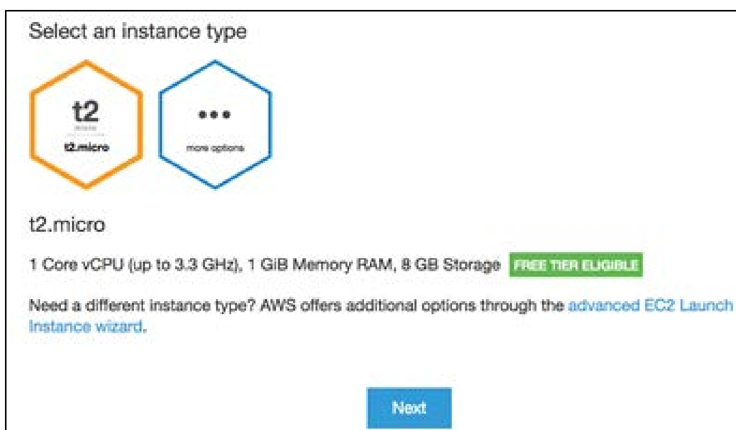



图 13-11 选择 Ubuntu 用于部署

(6) 根据硬件要求选择实例类型，如图 13-12 所示。

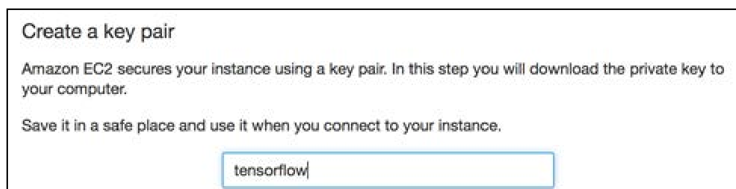



图 13-12

(7) 为安全起见，需要创建钥匙对以便登入机器。提供文件名，如图 13-13 所示。




图 13-13 确认机器名称

(8) 在图 13-13 中，你理应看见状态为 Completed（已完成）。现在，你可以通过单击如图 13-14 所示的按钮转换到 EC2 命令行。




图 13-14

(9) 启动实例会有一个短暂的停顿。当实例创建完成后，一个 IP 会显现出来以供登入，如图 13-15 所示。

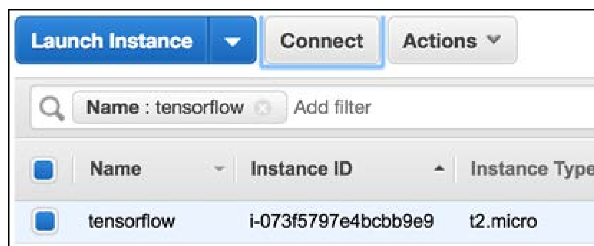



图　13-15

(10) 单击 Actions（操作）按钮，选择 Terminate（结束）可以在需要的时候结束实例，如图 13-16 所示。

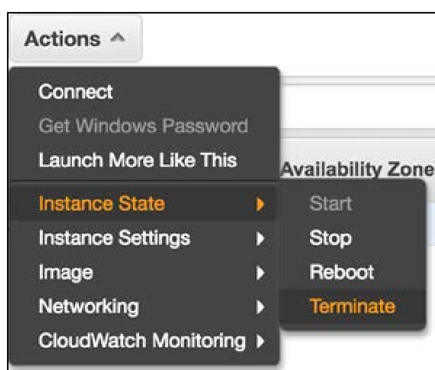



图　13-16

通过以上步骤可以根据需要的硬件要求创建实例并部署 TensorFlow Serving，用以推理客户的查询。

## 13.3.2　Google Cloud Platform

我们可以在 Google Cloud Platform（GCP）上新建实例并用其提供模型服务，就像在 AWS 上所做的那样。可以使用 Gmail 登录，进入 GCP。

(1) 单击如图 13-17 所示的按钮进入控制台。

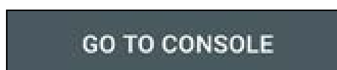



图　13-17

(2) 所有选项都可用于启动所需实例。在 Computer Engine（计算机引擎）下，选择 VM instances（VM 实例），如图 13-18 所示。

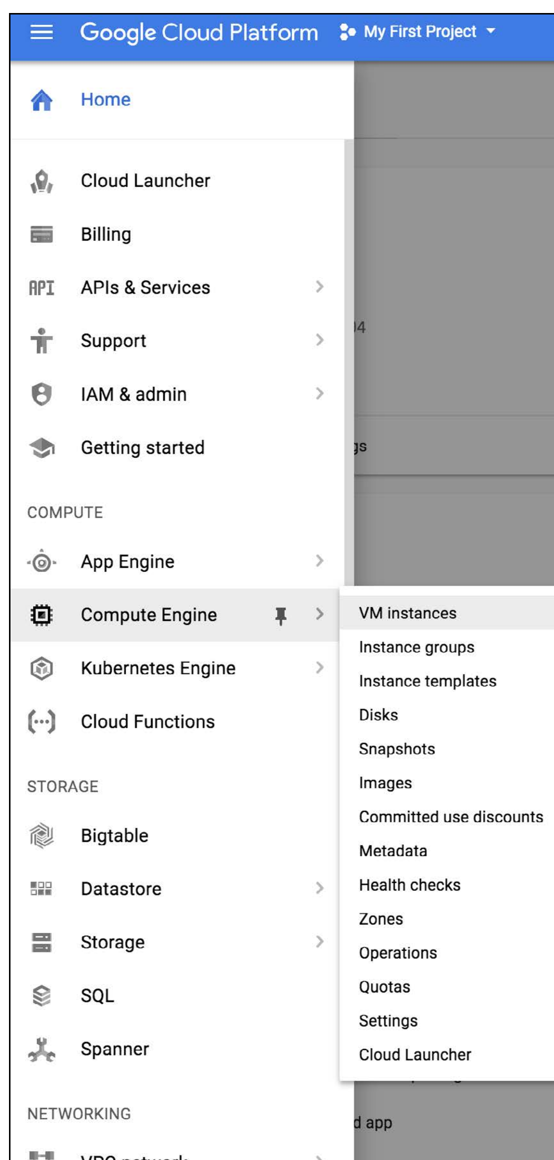



图 13-18　在控制台选择 VM 实例

(3) 接着点击 CREATE INSTANCE（创建实例），如图 13-19 所示。




图　13-19

(4) 现在可以指明所需核数和 CPU 了，如图 13-20 所示。

Create an instance

Name

instance-2

Zone

us-east1-b

Machine type

Customize to select cores, memory and GPUs.

2 vCPUs 7.5 GB memory Customize

Container

Deploy a container image to this VM instance. Learn more

Boot disk

New 10 GB standard persistent disk

Image

Ubuntu 16.04 LTS

Change

Identity and API access

Service account

Compute Engine default service account

Access scopes

Allow default access

Allow full access to all Cloud APIs

Set access for each API

Firewall

Add tags and firewall rules to allow specific network traffic from the Internet

Allow HTTP traffic

Allow HTTPS traffic

Management, disks, networking, SSH keys

You will be billed for this instance. Learn more

Create Cancel

图 13-20 选择对核、内存和 GPU 的需求

(5) 过一会儿，实例就可用了，如图 13-21 所示。

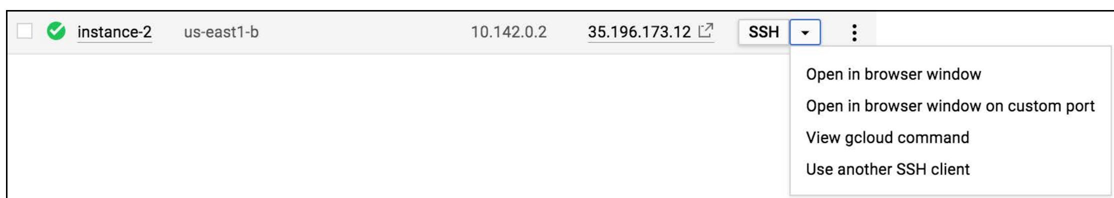



图 13-21

实例可以通过浏览器和 SSH 客户端访问，并用作模型服务。

## 13.4 在移动设备上部署

与云相比，在移动环境中部署模型具有各种优势，例如隐私保护和零延迟。iPhone 和 Android 等著名的移动平台提供了许多有助于将模型部署到移动环境的 API。

### 13.4.1 iPhone

Apple 公司为与机器学习相关的应用引入了 CoreML2，我们可以在其网站上找到详细信息。CoreML 有一个用于 NLP 的特殊框架，其中有用于分词、语言识别、POS 和命名实体识别等的预构建 API，我们也可以对自定义模型进行训练。在 Apple 设备中，使用 CoreML 模型的速度很快，且数据不必离开设备就可以进行推理。大多数 TensorFlow 模型可以转换为 CoreML 模型。

### 13.4.2 Android

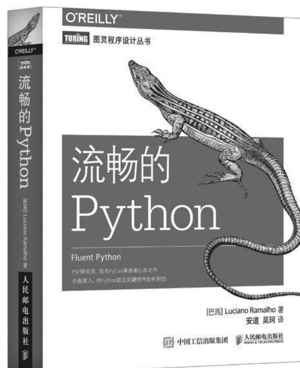
Android 应用无须更改即可直接使用 TensorFlow 模型。Android 中可以导出模型（类似于 TensorFlow Serving 中的导出）并在应用程序中使用。为了提高模型的效率，Graph Transform 工具提供了一组说明供参考。该工具可用于删除模型中用于训练的部分，从而减小模型的尺寸。

## 13.5 小结

本章研究了各种方法来为 NLP 任务部署训练好的模型。首先，我们学习了通过量化提高模型性能的方法，以及更快的推理方法。之后，我们了解了如何使用 TensorFlow Serving 部署模型以进行更快、可扩展的推理。接着，我们阐述了如何通过 AWS 和 GCP 进行云部署。最后，我们概述了某些移动平台上的部署。

最后一章介绍了如何部署训练好的模型并在云中提供服务。有了这些知识，你可以进一步探索如何将自己的模型部署到生产环境中去。

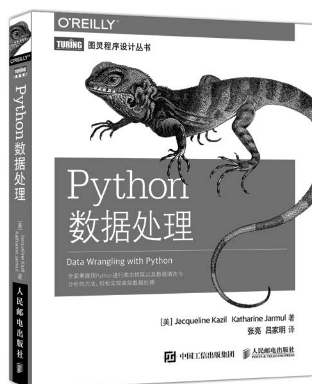
# 技术改变世界 · 阅读塑造人生



## 流畅的 Python

◆ PSF研究员、知名PyCon演讲者心血之作，Python核心开发人员担纲技术审校
◆ 全面深入，对Python语言关键特性剖析到位
◆ 大量详尽代码示例，并附有主题相关高质量参考文献
◆ 兼顾Python 3和Python 2

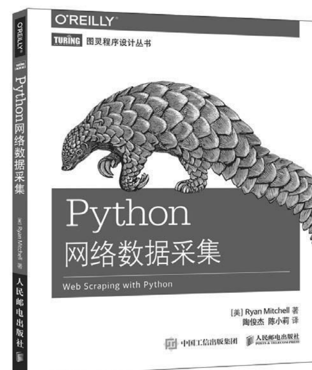
**书号：** 978-7-115-45415-7
**定价：** 139.00 元



## Python 数据处理

◆ 全面掌握用Python进行爬虫抓取以及数据清洗与分析的方法，轻松实现高效数据处理

**书号：** 978-7-115-45919-0
**定价：** 99.00 元



## Python 网络数据采集

◆ 全面展示网络数据采集常用手段，剖析网络表单安全措施
◆ 使用Python脚本和网络API一次性采集并处理海量网页数据，涵盖网络爬虫技术

**书号：** 978-7-115-41629-2
**定价：** 59.00 元

# Python自然语言处理实战

**Hands-on Natural Language Processing with Python**

自然语言处理（NLP）已在Web搜索、广告和客户服务等各个领域中得到广泛应用。借助深度学习，我们可以增强NLP在这些领域的性能。使用Python，可以利用深度学习模型执行各种NLP任务，以及应对当今的各种NLP挑战。

阅读本书后，你将学会将神经网络融入各种跨平台的语言应用程序中，使用NLTK和TensorFlow执行NLP任务并训练模型，以及通过强大的深度学习架构（例如CNN和RNN）增强NLP模型。

- 进行词语的语义嵌入以对实体进行分类和查找
- 通过训练将词语转换为向量，以执行算术运算
- 训练深度学习模型以检测推文和新闻的分类
- 使用搜索和RNN模型实现问答模型
- 使用CNN为各种文本分类数据集训练模型
- 实现深层生成模型WaveNet，以产生自然语音
- 将语音转换为文本并将文本转换为语音
- 使用DeepSpeech训练模型，将语音转换为文本

Packt>

图灵社区：iTuring.cn
分类建议：计算机 / 自然语言处理
人民邮电出版社网址：www.ptpress.com.cn







定价：59.00 元